RENSSELAER POLYTECHNIC INSTITUTE
KNOWLEDGE AND THOROUGHNESS
1824

Rensselaer Polytechnic Institute

Troy, New York 12181

RPI TECHNICAL REPORT MP-77

A PROPULSION AND STEERING CONTROL SYSTEM FOR THE MARS ROVER

by

Jeffrey M. Turner

A Study Supported by the

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

under

Grant NSG-7369

and by the

JET PROPULSION LABORATORY

under

Contract 954880

School of Engineering
Rensselaer Polytechnic Institute
Troy, New York

August 1980

Table of Contents

List of Figures

Abstract

This paper discusses the design of a propulsion and steering control system for the R.P.I. prototype autonomous Mars Roving Vehicle. The vehicle is propelled and steered by four independent electric motors. The control system must regulate the speeds of the motors so they work in unison during turns and on irregular terrain.

First, the paper presents an analysis of the motor coordination problem on irregular terrain, where each motor must supply a different torque at a different speed. A procedure is developed to match the output of each motor to the varying load.

Then, a design for the control system is given. The controller uses a microprocessor which interprets speed and steering commands from an off-board computer, and produces the appropriate drive voltages for the motors.

Acknowledgement

I would like to express my appreciation to everyone on the Mars Rover project for their efforts to make it a success. I would like to particularly thank those people who helped me with the propulsion and steering system:

Dr. Steven Yerazunis, for his enthusiasm and encouragement;

Dr. David Gisser, for his advice on the design, and assistance with this report;

Dr. Dean Frederick, for managing the project during a difficult period, and for his help with this report;

Graham Doig, and Karen Andersen, for wiring and documenting the electronics hardware;

Paul Dunn, for installing the invaluable M6800 Cross Assembler and Emulator package on the Prime computer;

my wife, Antoinette, for drawing the figures and proofreading this report.

1. Introduction

1.1. History

Since 1967, R.P.I. has studied various problems associated with an unmanned mission to Mars under grants from NASA and the Jet Propulsion Laboratory. Beginning in 1973, the primary objective of the study shifted to the construction of a prototype "Mars Roving Vehicle". The Mars rover would carry an on-board laboratory to study the surface of the planet at selected sites.

R.P.I.'s study focuses on the problem of obstacle detection and avoidance. The rover must move across unknown Martian terrain. Unfortunately, round trip radio communication between Earth and Mars will take about twenty minutes. Thus, direct radio control of the vehicle's motion from Earth is impractical, if not impossible. A hazard detection system, with an on-board computer, would identify dangerous terrain features, such as craters and boulders, and find a safe path around them.

By 1977, a half-scale prototype vehicle was completed. The prototype contained an on-board hazard detection system, and operated under control of an off-board computer via a radio link. Hazard detection was accomplished by firing a laser at the ground in selected azimuths along the vehicle's path. A laser detector was mounted below the laser and adjusted so that it could observe the laser spot only if the spot fell on terrain within safe height and slope limits. Thus, the detector classified each azimuth as a "safe" or "unsafe" direction[1]. This single-laser/single-detector system was demonstrated in laboratory and field tests in 1978.

In 1977, work began on the next generation hazard detector, called the elevation scanning system. This system fires laser shots in several azimuths at various elevations. A multiple element laser diode or CCD array will observe the spot, and give detailed information about the terrain ahead of the vehicle[2].

Since the completion of the single-laser/single-detector demonstrations, much work has been done to prepare the rover to carry the elevation scanning system. Some of the vehicle systems were directly affected by the change. However, other systems have been modified or redesigned to improve performance and correct problems which were identified during the laboratory and field tests. This report deals with the evaluation and redesign of the propulsion and steering control systems.

1.2. Propulsion and Steering Overview

The propulsion system must carry the rover over rough terrain as directed by the path selection algorithm, and supply important navigational information. The propulsion system consists of wheel motors, motor drivers, and motor controllers.

Originally, only the rear wheels of the vehicle were powered. A separate motor steered the rover by turning the entire front axle. The motors, controlled by relays, operated only in forward, reverse, or off modes. Motors were later added to the front wheels to help the rover climb steps.

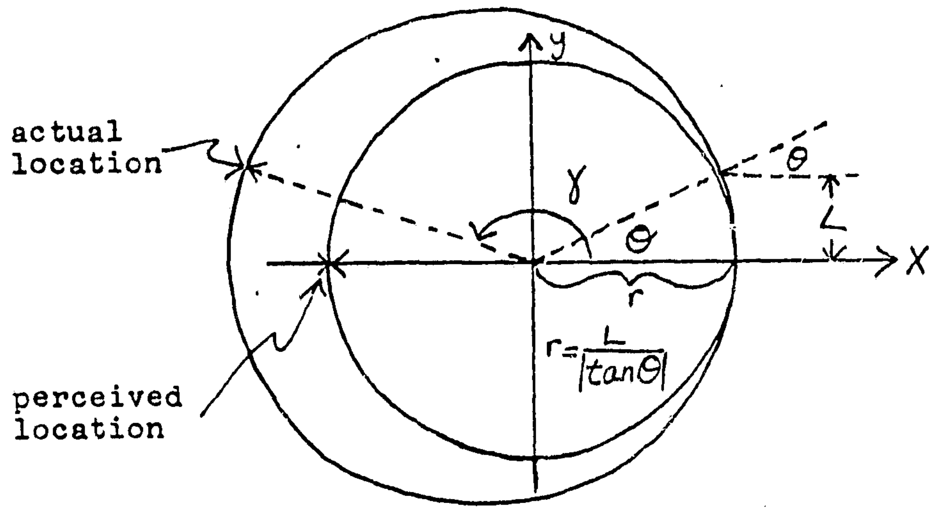
Unfortunately, the rover could not be steered when stopped and, since the wheel speeds were unregulated, it stretched and compressed under varying loads. For those reasons, a four wheel speed control was implemented[3]. All four wheel speeds were set to maintain the proper relationships during steering and turns on level ground. The steering motor still assisted in turning the front axle. This system served throughout laboratory testing of the single-laser/single-detector system.

Some problems remained with the four wheel speed control system. The vehicle still stretched, compressed, and lurched. The steering motor fought with the front drive motors during turns, and was removed. The wheel speeds needed frequent, difficult recalibration to keep them consistent with one another. More importantly, the computer could not reliably calculate the rover's true position from wheel speed and steering angle data. The new propulsion and steering control system should not have these faults.

1.3. Navigation and Performance Requirements

For safe navigation around an obstacle, the control computer must accurately and reliably know the position of the vehicle relative to the obstacle. Thus, the propulsion and steering system must at least provide accurate, consistent wheel speed, and steering angle information.

Although some measurement errors tend to cancel out over long distances, they generally will not cancel in the critical short range context. Errors in wheel speeds and steering angle have different effects. For instance, if we can tolerate a 2% uncertainy in position relative to an obstacle, we can allow wheel speed measurement errors up to 2%. Steering errors are not as simple, and cause greater problems. Consider the examples illustrated in figure 1-1. Here, the vehicle is making a constant turn, presumably to avoid a hazard that it has detected. For typical turns in



θ_o = perceived steering angle θ_1 = actual steering angle

r = turning radius L = vehicle length = 1.6 m

d = πr = distance traveled

Δd = error in navigation = $\sqrt{(\Delta x)^2 + (\Delta y)^2}$

where

$$\Delta x = 2\frac{L}{\tan\theta_o} - \frac{L}{\tan\theta_1}\left[1 - \cos\left(\pi\frac{\tan\theta_1}{\tan\theta_o}\right)\right]$$

$$\Delta y = \frac{L}{\tan\theta_1}\sin\left(\pi\frac{\tan\theta_1}{\tan\theta_o}\right)$$

Assume a 1° error in steering measurement

1) θ_o = -51.47° (4 increments left)
   θ_1 = -52.47° (2% error)
   d = 3.86 meters
   Δd = 0.17 meters (4.5% error)

2) θ_o = -38.6° (3 increments left)
   θ_1 = -39.6° (2.6% error)
   d = 6.28 meters
   Δd = 0.27 meters (4.3% error)

Figure 1-1
Steering Errors in Navigation

the four to six meter range, the relative position errors are much greater than the relative steering errors. In the example, a one degree (2%) steering error produces a 17 cm (4.5%) position error in about a four meter trajectory. The same 2% error in speed would produce only an 8 cm position error. Thus, we should pay particular attention to steering angle measurement.

One must also consider the performance of the vehicle on irregular terrain. Since individual motors drive all four wheels, the rover can climb moderate steps and slopes without much difficulty. However, a coordination problem exists with independent motors. Each must supply a different torque at a different speed as the rover turns and climbs. The control system should account for these differences so that the motors will work together, and not against one another.

2. Wheel Coordination

2.1. The Problem

When the rover moves forward on level ground, all four wheels should move at the same speed and deliver about the same torque. When the rover turns, or encounters rough ground, this is no longer true.

The rover steers by turning its entire front axle. This is called wagon steering. Obviously, the axle will turn if one of the front wheels moves faster than the other. However, all four wheel speeds must adjust if the rover is in motion while turning.

In a four-wheel drive automobile, there is one motor. A set of differentials allows each of the wheels to supply the necessary torque at the proper speed relative to the other three wheels. On the rover, there are four independent speed-controlled motors. If the control system can supply the proper set speeds for the particular terrain and steering angle, then speed feedback will insure that each motor supplies the necessary torque.

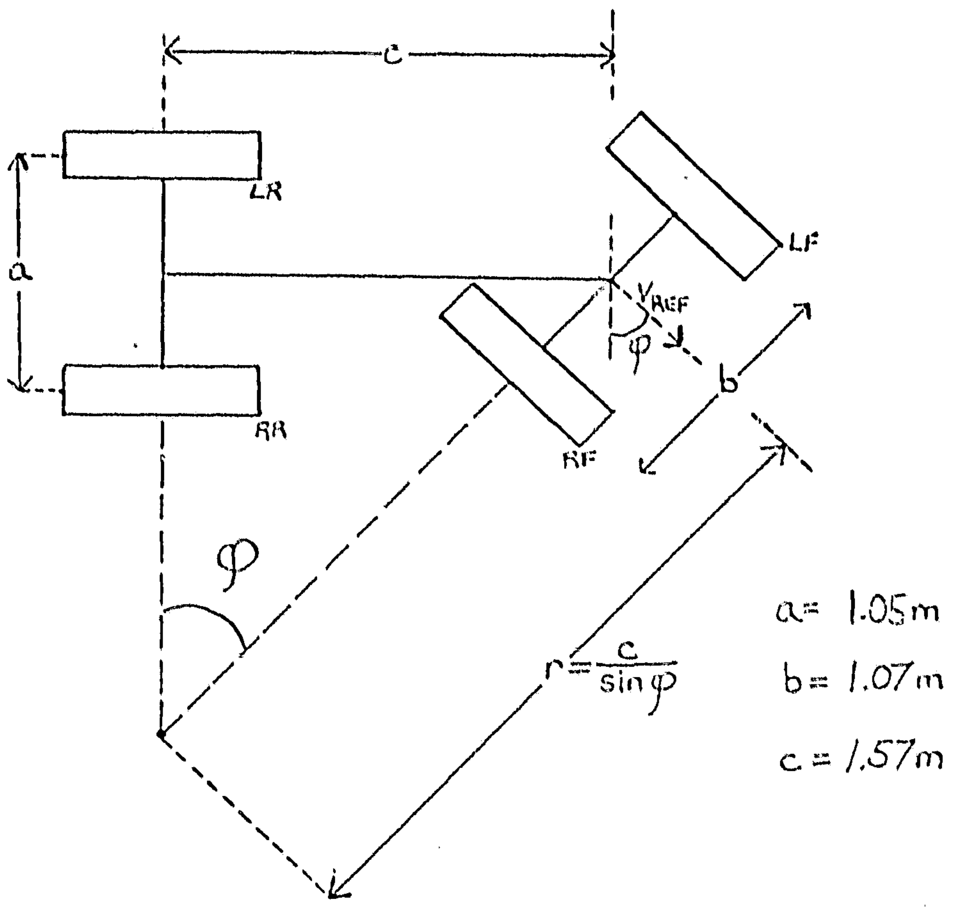
Tim Geis[3] developed the speed relationships as a function of steering angle on level ground. As the rover turns, the wheels move in concentric circles of different radii. The wheels with the longer turning radius must travel farther than the others. Therefore, they must turn faster. The relationship between wheel speeds is determined by the rover's geometry. Each wheel speed is normalized with respect to the speed of the center of the front axle. See figure 2-1.

On irregular terrain, the situation becomes more complex. Dave Knaub* studied the torque-speed requirements of a rigid two-wheeled vehicle climbing a slope. Again, the front-to-rear speed relationship is determined by geometry. For example, when the rover begins to climb a 30° slope the front wheel must turn 15% faster than the rear. Here, the local slopes under each wheel and the pitch angle are important. See figure 2-2. Torque requirements depend upon the vehicle weight and pitch angle. Torques are limited by slippage.

Knaub determined that there is a single acceptable speed ratio, and a range of acceptable torques at each point on the slope. He showed that unregulated motors, operating at a constant voltage, will adjust along their speed torque curves to an acceptable torque at acceptable speeds as the vehicle climbs the slope.

Should we, then, regulate the motors? If not, then which torques in the acceptable range should we choose?

Of course, we must regulate the motors. Recall that the motors did run without regulation in the original



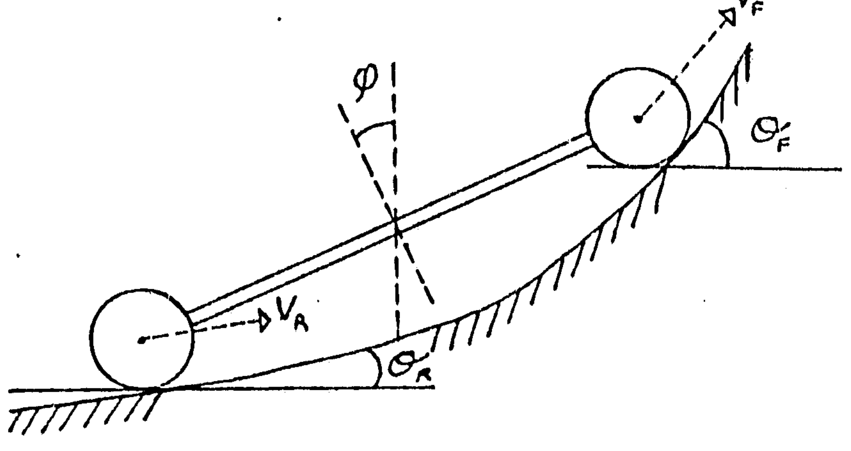
$$\frac{V_{RR}}{V_{REF}} = \cos\varphi - \frac{a}{2c}\sin\varphi$$

$$\frac{V_{LR}}{V_{REF}} = \cos\varphi + \frac{a}{2c}\sin\varphi$$

$$\frac{V_{RF}}{V_{REF}} = 1 - \frac{b}{2c}\sin\varphi$$

$$\frac{V_{LF}}{V_{REF}} = 1 + \frac{b}{2c}\sin\varphi$$

Figure 2-1
Wheel Speed Coordination on Level Ground



$$\frac{V_F}{V_R} = \frac{\cos(\theta_R - \varphi)}{\cos(\varphi - \theta_F)}$$

$$T_R \leq \mu_R R_N r \qquad T_F \leq \mu_F F_N r$$

where

V_F	= front wheel speed
V_R	= rear wheel speed
T_F	= front wheel torque
T_R	= rear wheel torque
μ_R, μ_F	= coefficients of friction
r	= wheel radius
R_N	= rear normal force
F_N	= front normal force
φ	= pitch angle
θ_R	= local slope of terrain under rear wheel
θ_F	= local slope of terrain under front wheel

R_N and F_N are functions of the vehicle weight, θ_R, θ_F, φ, and themselves[4].

Figure 2-2
Irregular Terrain Requirements

design. At that time, there were severe problems with maneuverability, and the vehicle compressed and stretched. The four wheel speed control system gave the vehicle much more maneuverability, and improved the stretching problem somewhat.

How, then, should we choose the torques? In a DC motor, there are three important parameters: input voltage, motor speed, and torque. A speed-torque curve exists for each input voltage. If we specify two parameters, then the third is also specified.

Suppose that the Knaub model is climbing a slope, as in figure 2-2. Input voltage and torque are specified. Obviously, both front and rear motors must supply more torque to match the load as the rover climbs. If the motors operate at a constant voltage, then the speed will drop. Unfortunately, it is not likely that the speeds of the front and rear motors will drop to the required front-to-rear ratio. The front and rear axles will tend to move toward or away from one another; since the vehicle is rigid, it supports a force between the axles and causes one of the motors slow down. The less the motor speed varies with applied torque, the greater this force must be to load the motor down. The load is not distributed evenly between the wheels.

If, instead of voltage and torque, we could choose the speed and torque that the motor produces, then we could adjust the mismatch force between the axles. If we can specify the wheel set speeds a priori to meet the proper geometric ratio, then the torques can meet the load requirement without a net front-to-rear force.

There is no simple way to a priori specify the proper set speeds on irregular terrain. It would be necessary to measure the local slopes under each of the four wheels, as well as the pitch, roll, and steering angles. Even if all of these variables were measured, and the motor drivers carefully calibrated to eliminate mismatch, the algorithm would be complex. A feedback solution seems more promising.

2.2. Strain Feedback

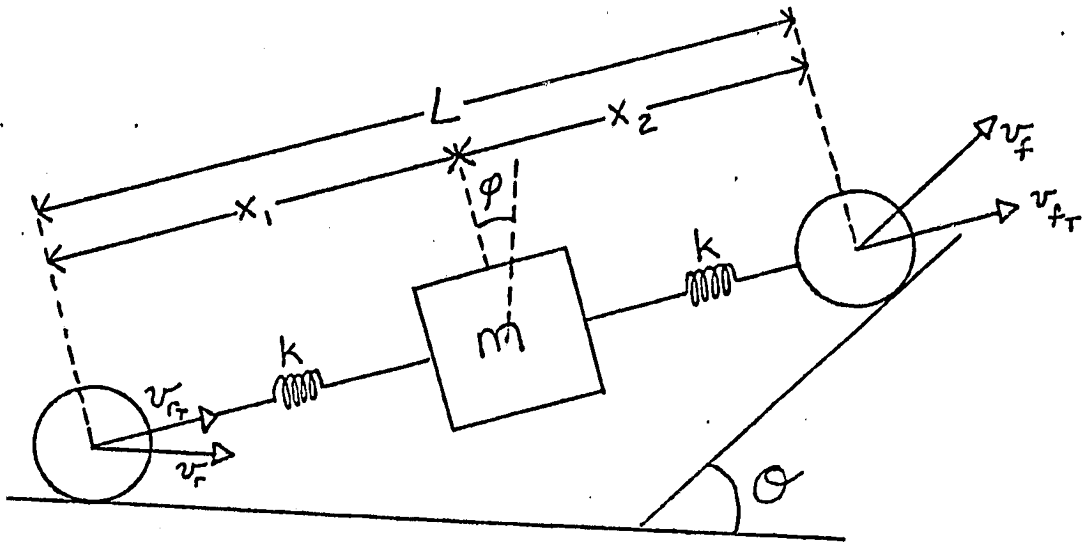
We desire to adjust the torques and speeds to acceptable relationships and distribute the load rationally between the four motors. A wheel speed mismatch adjusts itself by changing the load distribution. Since one motor carries most of the load, the vehicle's climbing ability is reduced. In fact, the motors may even work against one another. If we can detect the conditions under which the wheel speeds adjust themselves, we can choose an input voltage which will result in the proper speeds and torques.

First, assume that there is no mass and the vehicle is moving on level ground toward a 30° slope. As soon as the front wheel begins to climb the slope, geometry dictates that it should turn 15% faster than the rear wheel. Since the front of the vehicle is climbing too slowly, the rear will push the front. If the motors operate at constant voltage, or with a speed control, the front will resist. The motors will expend useful climbing power crushing the vehicle between them.

We can, however, measure this force with a strain transducer, and apply enough power to the front wheel so that it will turn faster, and carry its share of the load. Alternately, we could slow the rear wheel down, with the same results. Under these conditions, there will be no front-to-rear force.

Consider the simple model in figure 2-3. A two-wheeled vehicle climbs a slope with identical springs suspending the vehicle's mass between the wheels. The force of gravity causes a stretch in one spring, but an equal compression in the other. A set speed error would cause a net stretch or compression in both springs. Thus, it is easy to isolate the propulsion system error from the gravitational effect.

Figure 2-4 shows a more realistic model of the RPI

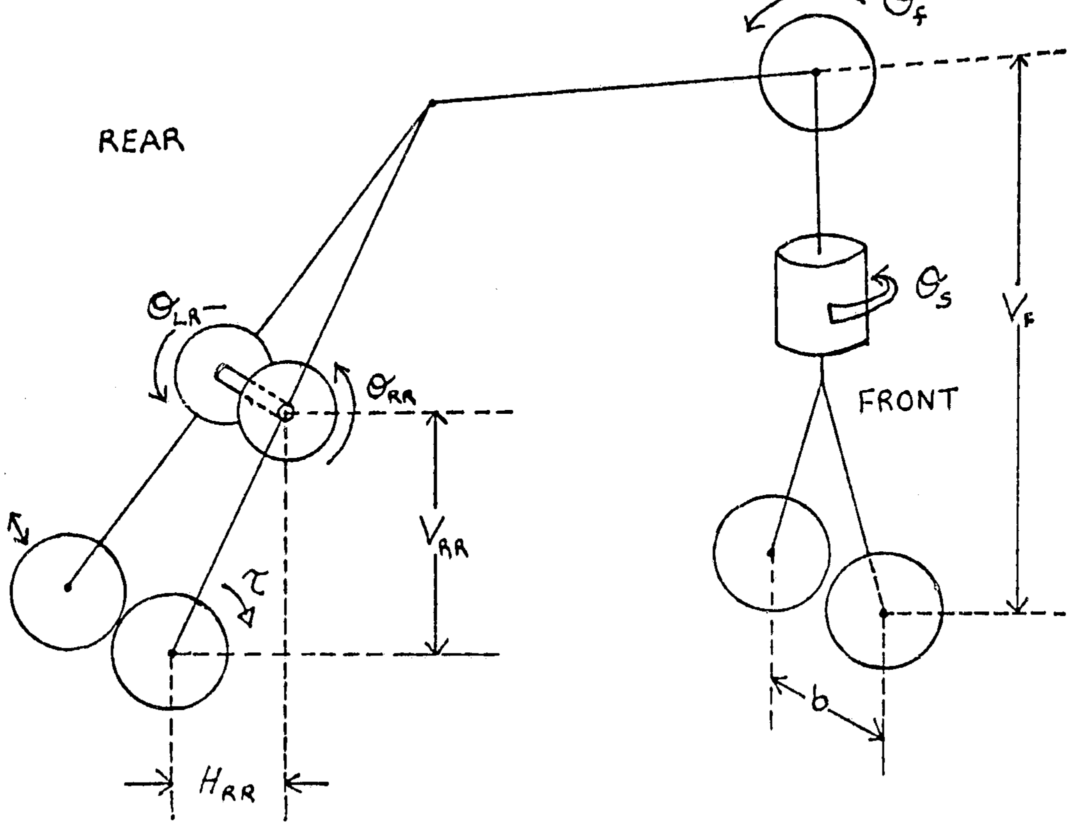


$$x_1 = \tfrac{1}{2}\left(L - \frac{Mg}{K}\,\mathrm{Sin}\,\varphi\right)$$
$$x_2 = \tfrac{1}{2}\left(L + \frac{Mg}{K}\,\mathrm{Sin}\,\varphi\right)$$
$$\frac{d}{dt}\,L = v_{f_T} - v_{r_T}$$

where

x_1	=	the stretch in the rear spring
x_2	=	the stretch in the front spring
L	=	the length of the vehicle
M	=	the mass of the vehicle
g	=	gravitational constant
φ	=	pitch angle
v_{f_T}	=	front wheel speed, parallel to vehicle
v_{r_T}	=	rear wheel speed, parallel to vehicle

Figure 2-3
Ideal Spring Model



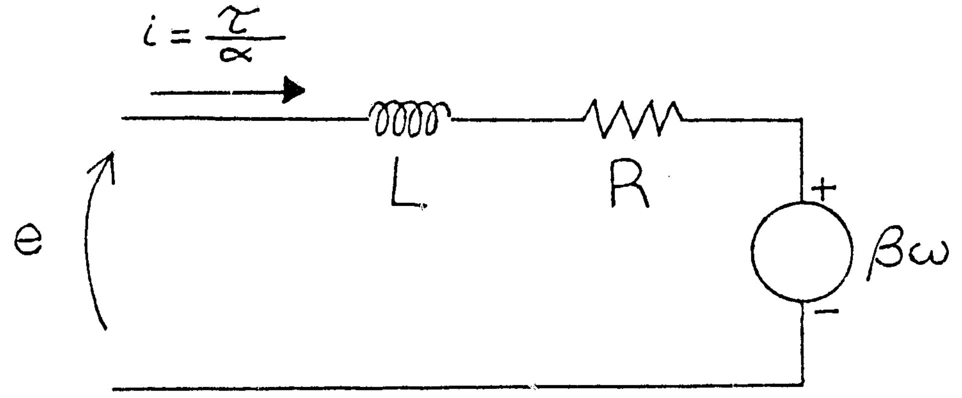


Figure 2-4a
Torsion Bar Model

Front torsion bar displacement, θ_f:

$$J_f \ddot{\theta}_f + B_f \dot{\theta}_f + K_f \theta_f = V_f\, r\, (\tau_{rf} + \tau_{lf}) \cos \theta_s$$

Left rear torsion bar displacement, θ_{lr}

$$J_r \ddot{\theta}_{lr} + B_r \dot{\theta}_{lr} + K_r \theta_{lr} = r\, \tau_{lr} V_{lr} - W_{lr} H_{lr}$$

Right rear torsion bar displacement, θ_{rr}

$$J_r \ddot{\theta}_{rr} + B_r \dot{\theta}_{rr} + K_r \theta_{rr} = r\, \tau_{rr} V_{rr} - W_{rr} H_{rr}$$

Steering angle, θ_s:

$$J_s \ddot{\theta}_s + B_s \dot{\theta}_s = \frac{br}{2} (\tau_{lf} - \tau_{rf})$$

for each motor:

$$\frac{d}{dt}\tau = \frac{\alpha}{L} \left(e - \beta\omega - \frac{R}{\alpha}\tau \right)$$

where

J	is the moment of inertia of each strut
B	is the viscous friction
K	is the spring constant of each torsion bar
r	is the wheel radius
V	is the vertical distance from the axle to the torsion bar axis
H	is the horizontal distance from the axle to the torsion bar axis
W	is the vehicle weight on a wheel
τ	is the motor torque
R	is the motor armature resistance
L	is the motor's inductance
β	is the back EMF constant
α	is the torque constant
ω	is the motor speed
e	is the motor input voltage
i	is the motor current
b	is the distance between the front wheels

Figure 2-4b
Torsion Bar Model Equations

rover. Instead of spring displacements, we measure strains in torsion bars in the front and rear. The payload is supported between these torsion bars. Since the bars supporting the rear struts are independent, and since we measure the steering angle, it is possible to adjust the voltage to each motor so all wheels work in unison. We correct for errors between the left and right front wheels by measuring the steering angle. We correct for net front-to-rear errors by observing the relative strains in the front and rear torsion bars. Finally, we correct for errors between the left and right rear wheels by measuring the relative strains in the rear torsion bars.

While this approach seems attractive, it is not so easy to implement. On the real rover, the stress measurements are not straightforward. Most of the vehicle strain does not actually occur in the torsion bars, but in the wheels and struts. Even if the strains were accurately measured, it is unlikely that the strains due to propulsion errors could be reliably isolated from strains due to vehicle weight. The correction algorithm would be very complex. This approach might warrant further study, but there is a simpler and more promising alternative.

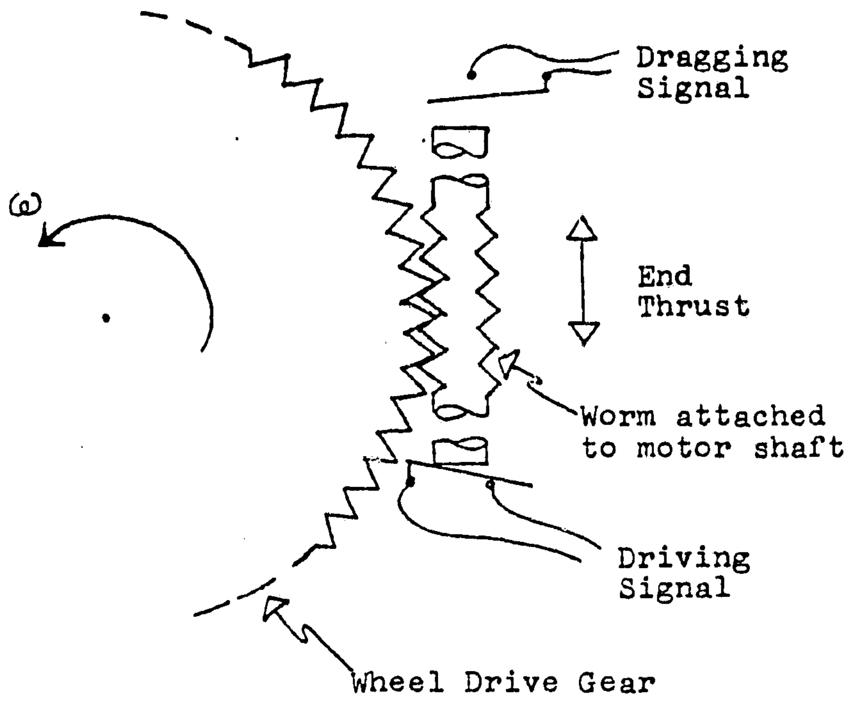
2.3. Mismatch Feedback

As the rover climbs over irregular terrain, various torque and speed relationships hold for each wheel. When the input voltages to the motors do not allow the motors to meet the relationships, power is wasted and the rover is unnecessarily strained. This extra strain results because some motors push the others.

Reconsider the two-wheeled vehicle just beginning to climb a 30° slope. The rear wheel pushes the front wheel. The torque required from the rear motor increases, and the rear slows. The load on the front motor is reduced, so it speeds up. As climbing continues, a compressive force develops on the vehicle. This compressive force indicates that the front wheel is dragging. In other words, the rear wheel is actually forcing the front wheel to turn.

On the R.P.I. rover, the wheels are driven by a worm gear arrangement. Thus, the front wheel cannot be forced to turn much faster than the front motor drives it. The extra loading on the rear wheel must slow it to 85% of the front wheel speed. Nevertheless, the compressive force developed by the rear motor still TRIES to turn the front wheel faster.

Refer to the sketch of a worm gear in figure 2-5. Normally, the motor drives the wheel, and there is a downward end-thrust on the worm. This end-thrust is



End Thrust Down	$\Rightarrow$	Motor Driving Wheel
End Thrust Up	$\Rightarrow$	Wheel Dragging

Figure 2-5
Mismatch Sensors

produced when the wheel resists the motor. If, however, the other wheel tries to turn it, then it does not resist. In fact, it pulls the worm gear upward. The reversal of the end-thrust indicates that the motor is no longer powering the wheel. This is a direct indication of a mismatch between the wheels.

If the control system could detect the end-thrust reversal, it could supply more power to the dragging motor and thereby increase the torque and speed output. The wheel speeds could meet the necessary torque and speed relationships; the load would be rationally distributed between the motors. At very worst, the motors would no longer fight one another, so they would no longer waste power compressing or stretching the rover.

Fortunately, it is not difficult to detect the reversal. By allowing the worm to move slightly, ordinary switches on the worm shaft can detect the direction of the end-thrust. The switches provide all the necessary information in digital form, and do not have problems with calibration or drift.

A computer program has been written to simulate the rover and test this idea. The simulation assumes the two-wheeled rover shown in figure 2-3. The front and rear are connected by an ideal spring which suspends the payload. Each motor is speed-controlled. The controller uses the wheel mismatch information to modify the rear set speeds while the front set speeds are not changed. Figures 2-6 and 2-7 show the equations used and a block diagram of the model control loop. The model parameters, given in figure 2-6, were estimated from measurements of the R.P.I. rover.

The controller allows the wheel speeds to seek whatever speed geometry dictates for them. Whenever the switches indicate that one wheel is dragging, the wheel set speed is increased (or decreased) by an arbitrary amount. This new set speed is low-pass filtered, with a time constant similar to the rover's time constant. When the actual wheel speed reaches the speed required by the geometry of the rover on the terrain, the motor again begins to drive the wheel. The switch turns off. The unfiltered set speed returns to its old value, and so the set speed begins to decrease (or increase). Eventually, the wheel will again begin to drag and the process repeats. There will be some small speed oscillation, but the resulting wheel speeds should follow the required speeds rather closely. In any event, the wheels will not fight one another.

Figures 2-8 through 2-10 compare the model rover's performance with and without the worm-gear sensors. In the figures, the rover climbs a ramp. The scales are the same. Without the sensors, the motors fight constantly; the rover

Mismatch Compensator Model

Use Ideal Spring Model, Figure 2-3
Wheel Mismatch Sensors, Figure 2-5

cumulative speed mismatch:

$$\Delta = \int \left\{ \omega_f \frac{\cos(\phi - \theta_f)}{\cos(\phi - \theta_r)} - \omega_r \right\} dt$$

Translation of worm gear:

$$T = \begin{cases} \Delta & \Delta < T_{max} \\ T_{max} & \Delta \geq T_{max} \end{cases}$$

Stress in vehicle spring:

$$S = \begin{cases} 0 & \Delta < T_{max} \\ K(\Delta - T_{max}) & \Delta \geq T_{max} \end{cases}$$

load torque on motor (always positive, since the wheel cannot turn the motor through the worm gear)

$$\tau_{load} = \begin{cases} \text{MAX}(W \sin\phi - S,\ 0) & \text{for rear} \\ \text{MAX}(W \sin\phi + S,\ 0) & \text{for front} \end{cases}$$

motor speed:

$$\omega = \frac{1}{J}\left(\frac{\alpha}{R} e - \tau_{load}\right) - \frac{\alpha\beta}{JR}\omega$$

α = motor torque constant, 0.8 Kg·m/amp
β = back EMF constant, 12 V/rad/sec
R = motor armature resistance, 2.7 ohm
J = effective moment of inertia on each wheel
W = weight, (half of rover), 35 Kg

Figure 2-6

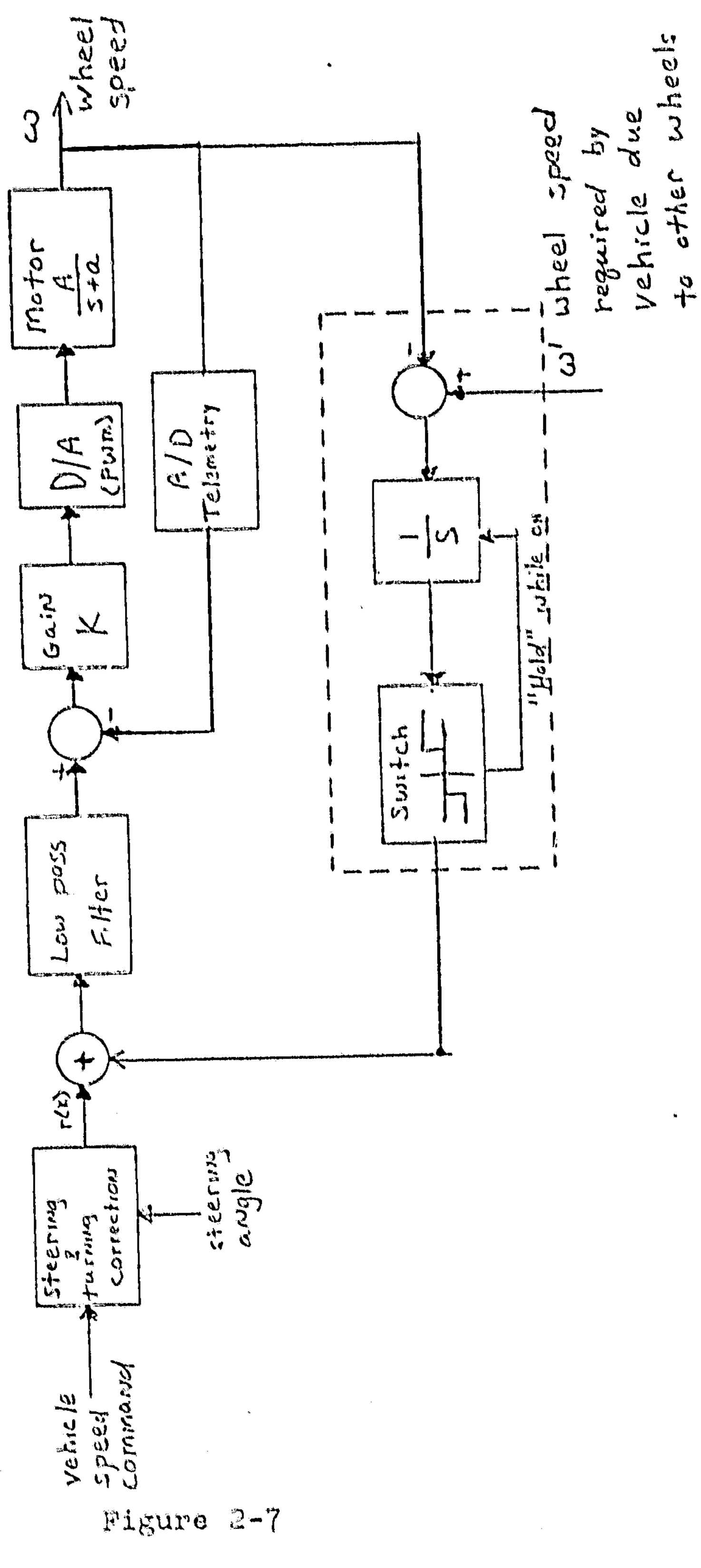
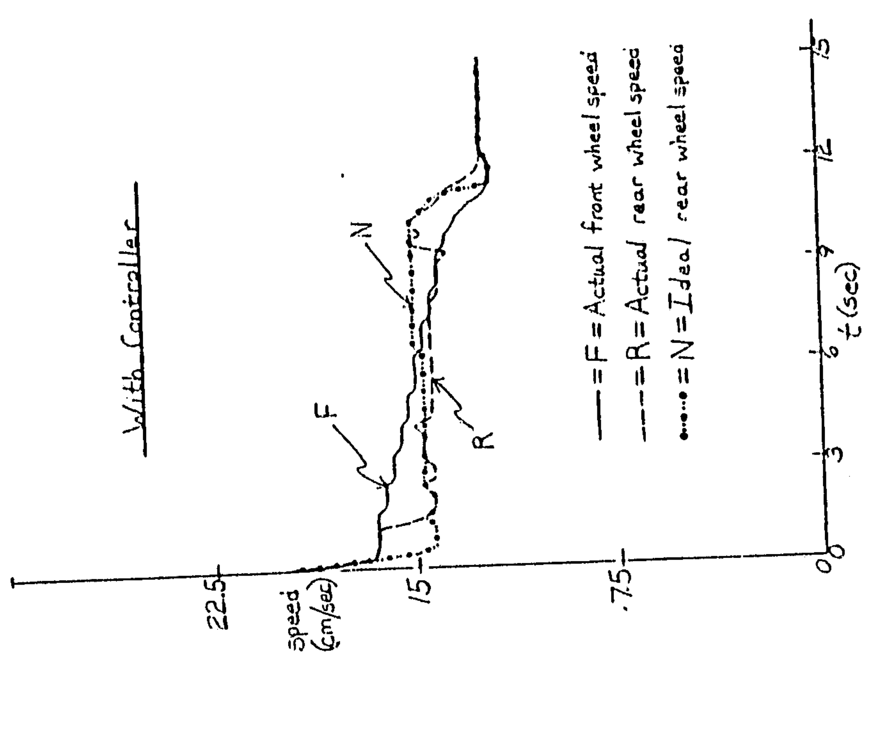


Figure 2-7

Mismatch Compensator Model Block Diagram



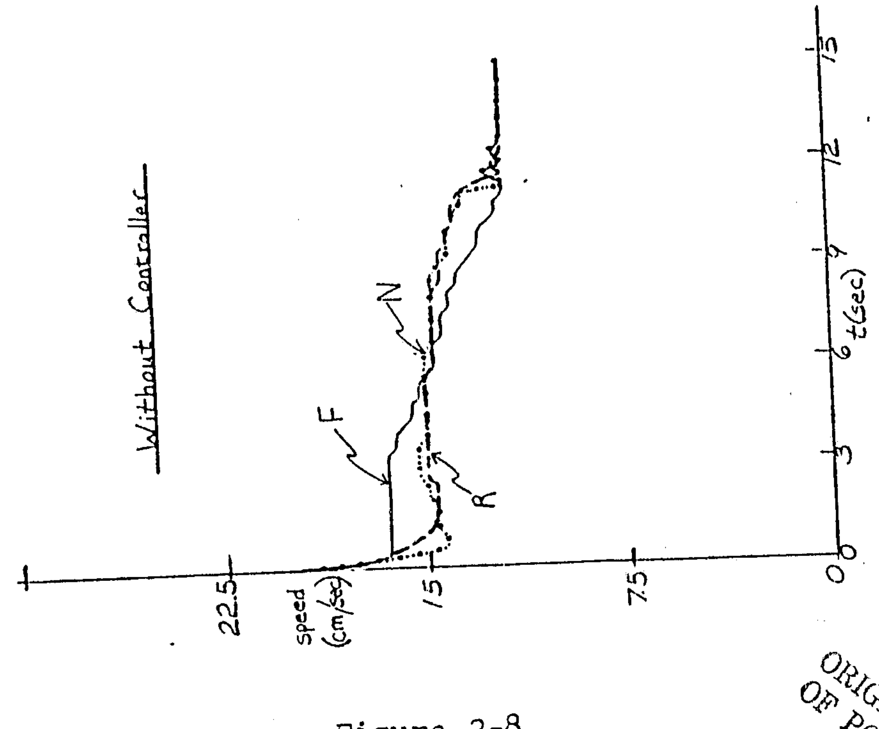
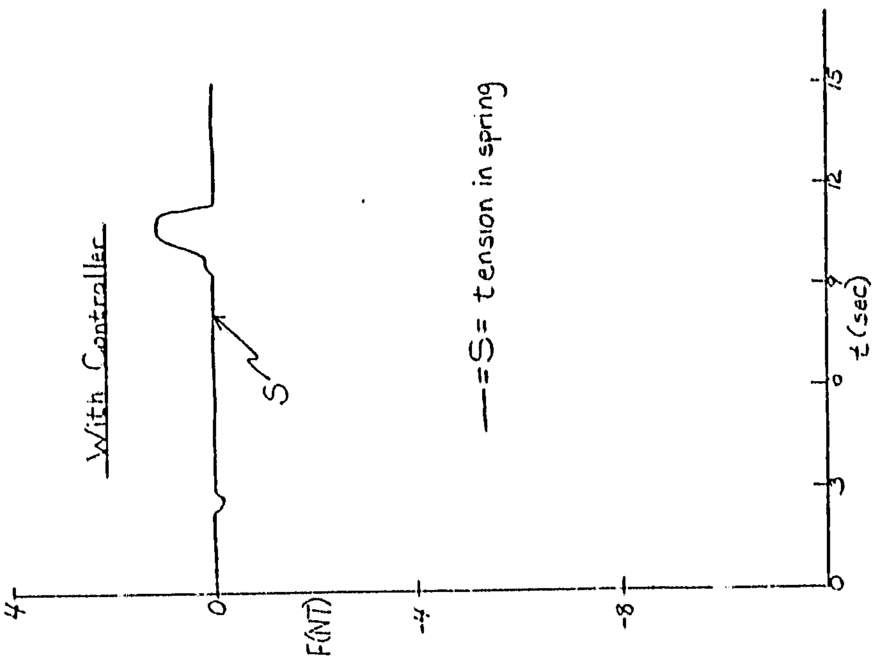


Figure 2-8
Simulated Wheel Speed Comparison



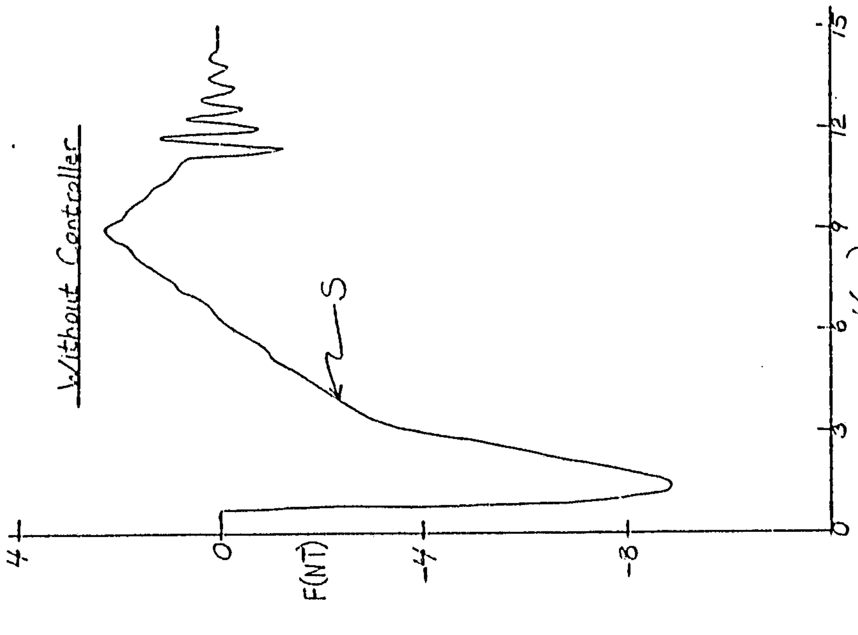


Figure 2-9
Simulated Vehicle Tension Comparison

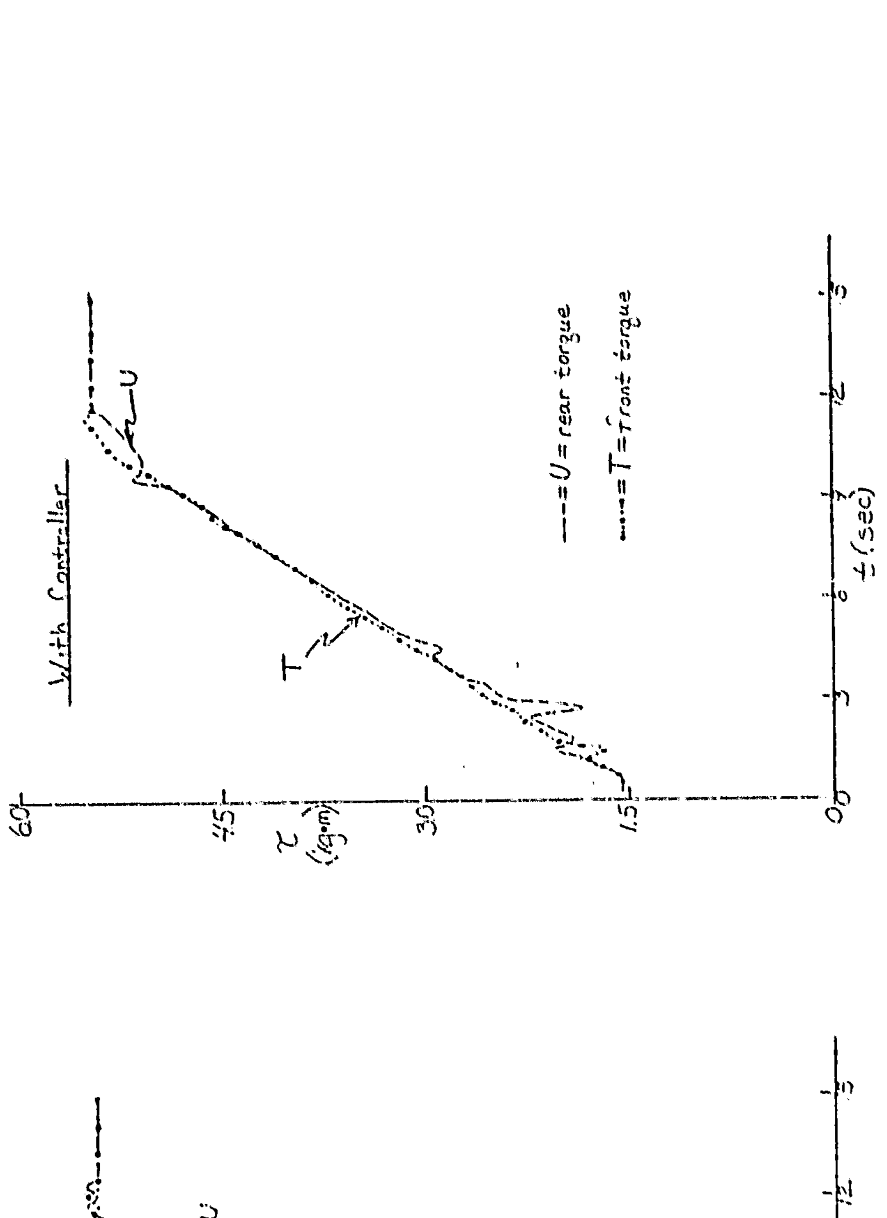


Figure 2-10
Simulated Motor Torque Comparison

stretches and compresses. With the sensors, the motors fight only while the controller reacts to the mismatch signal. The stretch and compression peaks during this time reach only one-sixth of the peaks without the sensors. The wheel speeds follow the required wheel speeds, and the torques produced by each wheel are the same.

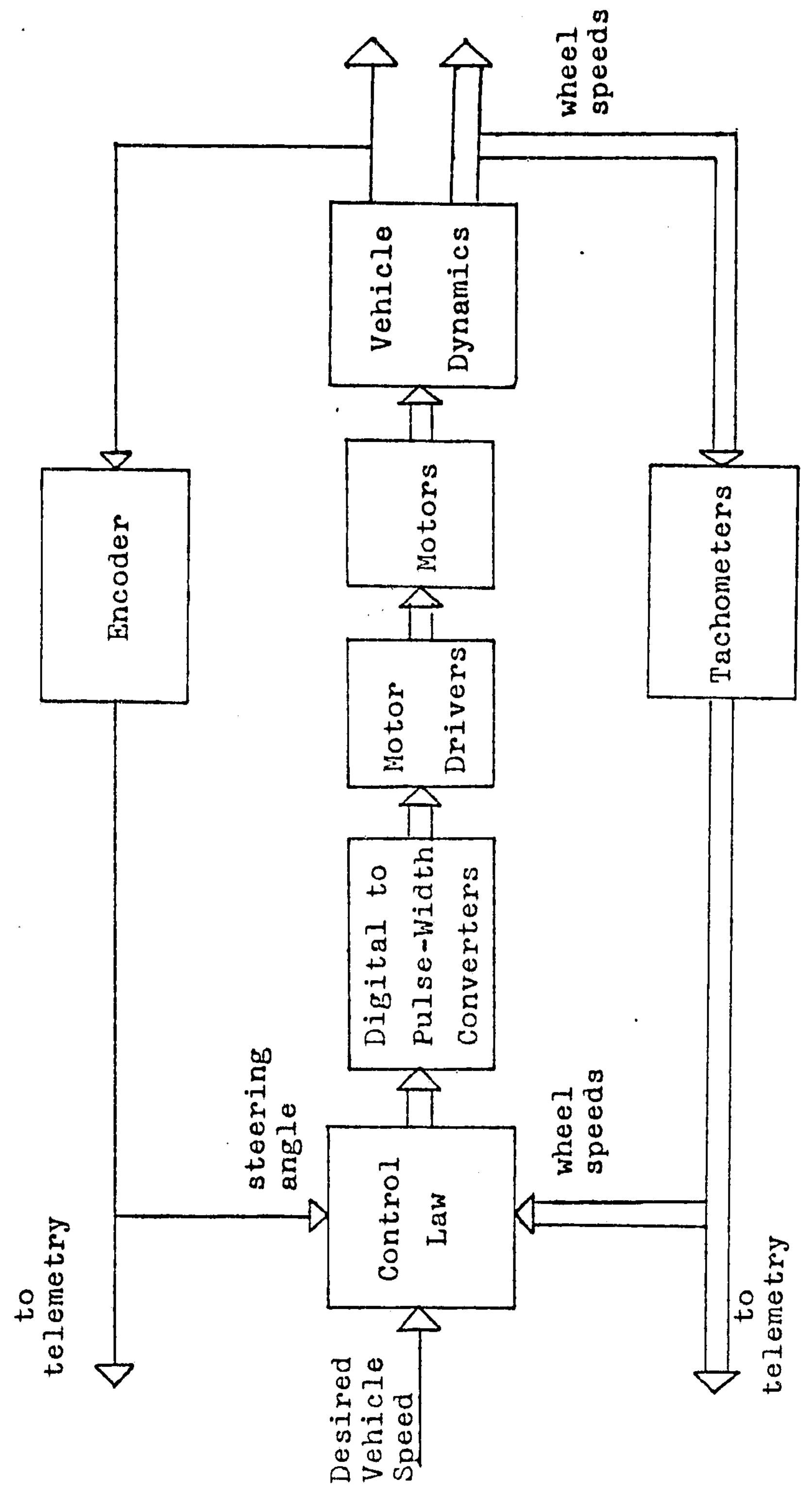
A controller utilizing sensors on the worm gears can, then, properly regulate the motors on irregular terrain. It will also compensate for set speed errors and mismatched motor drivers since it responds to any kind of torque or speed error. We are now implementing and testing this system on the actual rover. It should greatly improve the rover's performance on all terrain.

3. Implementation

3.1. System Overview

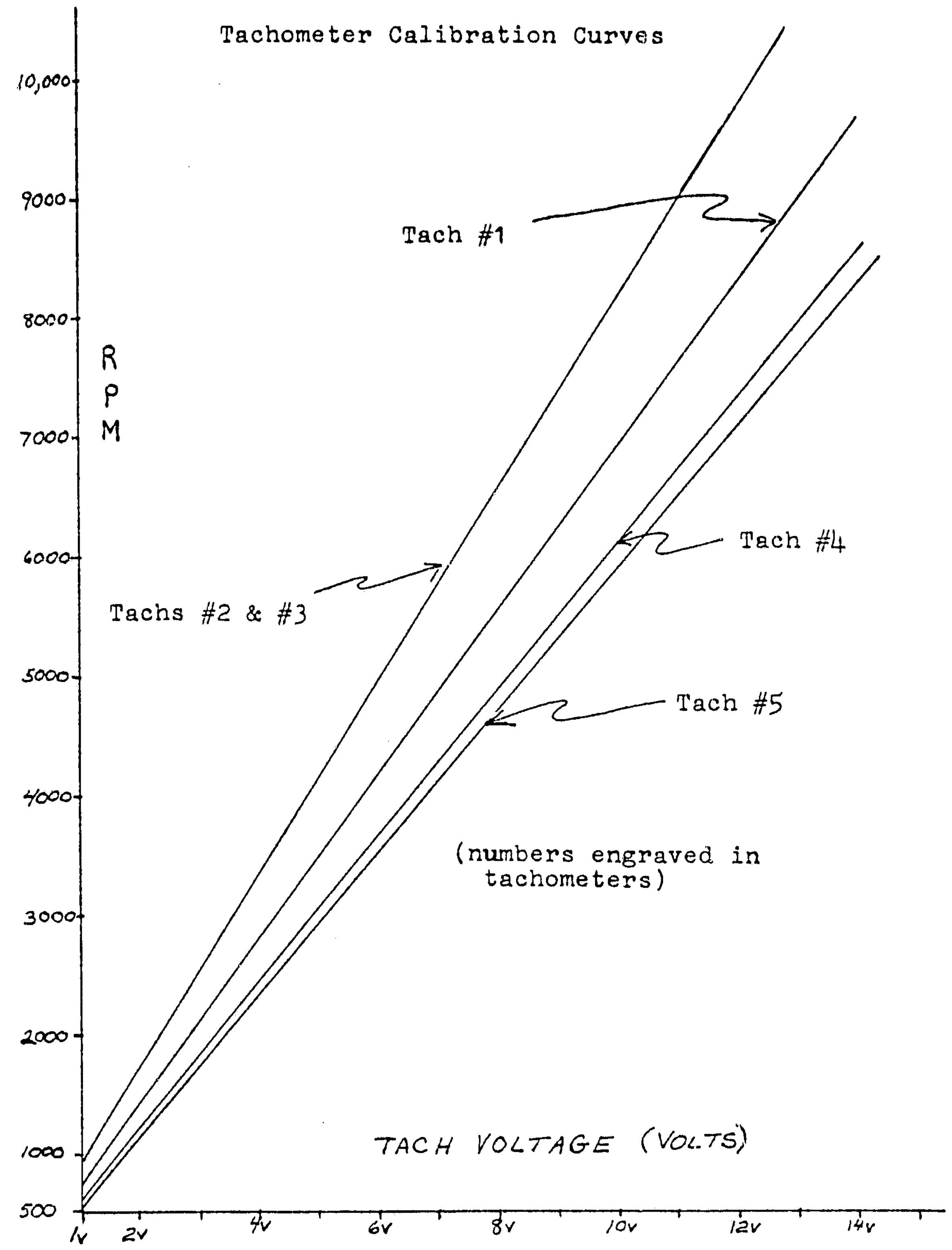
The propulsion and steering control system must obtain the steering angle, wheel speeds, mismatch data, and commands from the operator or off-board computer. It must supply appropriate voltages to each motor so that each wheel will maintain the correct torque and speed. See figure 3-1.

Steering angle and wheel speed measurements are also required by the off-board computer for path selection. The measurements must be accurate, and the equipment must be easy to calibrate. The steering angle measurement is critical. We needed an 8-bit encoder, with 1.4° resolution, to meet navigational requirements. I chose a 10-bit optical shaft encoder (Litton model 76NB10[illegible]E1), which has a resolution of 0.35°, since it was available at the same price. It is possible that a precision potentiometer would have provided adequate resolution, but additional errors and calibration adjustments are introduced by buffers and A/D conversion. The wheel speed measurements need not be as precise, so I decided to use the tachometers which were already on the vehicle. We tested the tachometers and found them to be fairly linear, but each must be calibrated separately. See figure 3-2. If more accuracy is required



Propulsion and Steering System Block Diagram

Figure 3-1



Tachometer Calibration Curves
Tach #1
Tachs #2 & #3
Tach #4
Tach #5
(numbers engraved in tachometers)
R P M
TACH VOLTAGE (VOLTS)
10,000
9000
8000
7000
6000
5000
4000
3000
2000
1000
500
1v
2v
4v
6v
8v
10v
12v
14v

Figure 3-2

at a later time, these can be replaced by optical tachometers.

The on-board control system must interface with these transducers and perform all the necessary control calculations. A microprocessor-based controller is the best choice.

If the feedback control was done by analog circuitry, the output from the tachometers can be used directly. However, two D/A converters would be required to interface with the command link and steering encoder. A potentiometer could be used to measure the steering angle in conjunction with the encoder to eliminate one of the D/A converters. Even then, there is a problem correcting the wheel speeds during turns. The correction is a complicated geometric function of the steering angle and vehicle dimensions. The output of the encoder must address a read-only memory containing the correction factor, and a multiplying D/A converter must scale the analog set speed. We could rely upon mismatch feedback from the worm-gear switches to replace precise steering correction, but the wide range of speeds is a problem for the correction algorithm. Steering information must at least tell the mismatch controller which direction the wheels should turn. Altogether, the hardware would be complicated and difficult to calibrate.

A hard-wired digital controller should be easier to calibrate, but would still be complex. It could interface directly with the steering encoder and command link without any conversion. The telemetry system could do the A/D conversion required to supply the wheel speeds. Unfortunately, the low pass filters, steering correction, and proportional speed control are awkward to build.

A microprocessor shares the advantages of both hard-wired approaches, but few of the disadvantages. It is easy to interface to all of the existing systems. More importantly, it offers flexibility. The control algorithms could be modified, expanded, or completely replaced with the addition of little or no hardware. A user interface can be provided for laboratory or field calibration. It can perform a wide variety of control tasks on the rover, besides steering and propulsion. A Motorola M6800 microprocessor system was chosen since there was so much technical support for it on campus.

3.2. Hardware Description

The heart of the propulsion and steering control system is a Motorola stand-alone microcomputer board. An additional board contains the interface to the telemetry system, to motor driver circuitry, and to the parameter selection switches. These boards, plus the telemetry system, are contained in a single card cage. Motor driver circuitry is mounted on each strut.

3.2.1. Microprocessor Board

The propulsion control system uses the Motorola M68SAC1 stand-alone microcomputer board^5. The M68SAC1 was intended for use with a Motorola development system. In addition to the microprocessor, the board contains four parallel I/O ports (PIA's), two asynchronous serial I/O ports (ACIA's), 348 bytes of random access memory (RAM), and 3K bytes of programmable read-only memory (EPROM). It can be configured in a variety of ways by switches and jumpers. Appendix 6.2 gives the list of jumpers and switch settings that are needed to use the microcomputer in the propulsion system.

One of the serial ports, U12, acts as the interface to the command link. It interrupts the processor whenever a command is received. The other serial port, U4, is connected to an RS-232 interface and so can be connected to a terminal. The propulsion system uses U4 to display the wheel speeds and and steering angle. If any character is received from the terminal, U4 issues a non-maskable interrupt which stops execution of the propulsion system and transfers control to the MINIBUG II software. The command link port operates at 110 baud, and the terminal port operates at 9600 baud.

Obviously, the board has capabilities beyond propulsion and steering control. Only two other control functions remain on the current rover: strut control, and directional gyro control. If the microprocessor performs these functions, none of the old rover electronics will remain. This is an advantage, since an additional card cage must be built if any of the old electronics are kept. The software to perform these functions has been written. It uses only one of the parallel ports (U5B). The other three parallel ports are available for future expansion. Appendix 6.2 discusses the additional hardware required to drive the struts and gyro.

3.2.2. Telemetry Interface

The microcomputer must have current information about the rover in digital form. For propulsion and steering control, it needs all four wheel speeds, the steering angle, and wheel mismatch data. It needs the gyro reading for directional gyro control.

The off-board computer also needs this information. Therefore, a telemetry system samples each important vehicle state and transmits it to the off-board computer via a radio link. The telemetry system contains a data multiplexer which presents each 16-bit data word, in turn, to the transmitter with an identifying address. For the old telemetry system[6], the vehicle states were measured by analog devices: potentiometers and tachometers. A 16 channel analog multiplexer delivered each measurement to a 12-bit A/D converter. Four address bits were required to identify each of the 16 possible vehicle words. The new telemetry system, currently under construction[7], provides 16 analog and 16 digital channels. Thus, there is a five bit address.

In either case, the data multiplexer presents a parallel 16-bit data word to the transmitter for serialization. Rather than duplicate the multiplexer circuitry, the microcomputer obtains its information from the telemetry system's data multiplexer through the telemetry interface. The interface also allows the microprocessor to send data to the off-board computer.

The interface is a block of random access memory (RAM) which can accept data from either the telemetry system or from the microcomputer. See figure 3-3. The RAM appears as 128 16-bit words to telemetry, and 258 8-bit words to the microprocessor. Either system can obtain sole read and write access to the RAM.

The telemetry system requests access, via the signal "Telemetry Data Request" (TDR), whenever there is valid data at the output of the multiplexer, or whenever the multiplexer wishes to take data from the RAM during its polling sequence. The data multiplexer gives the same address to identify the word to the RAM and to the transmitter. It indicates whether it will read from or write to the RAM with the signal "Telemetry Read/Write". Unless the microprocessor already has sole access, the RAM controller returns the signal "Telemetry Data Grant" (TDG). The controller transfers the 16-bit word to or from the specified address.

The telemetry system must use the TDG and Telemetry Read/Write signals only if the polling sequence includes data from the microprocessor. Telemetry Read/Write should be grounded if it is not used. Naturally, it is up to the telemetry system to ignore data from the RAM if it

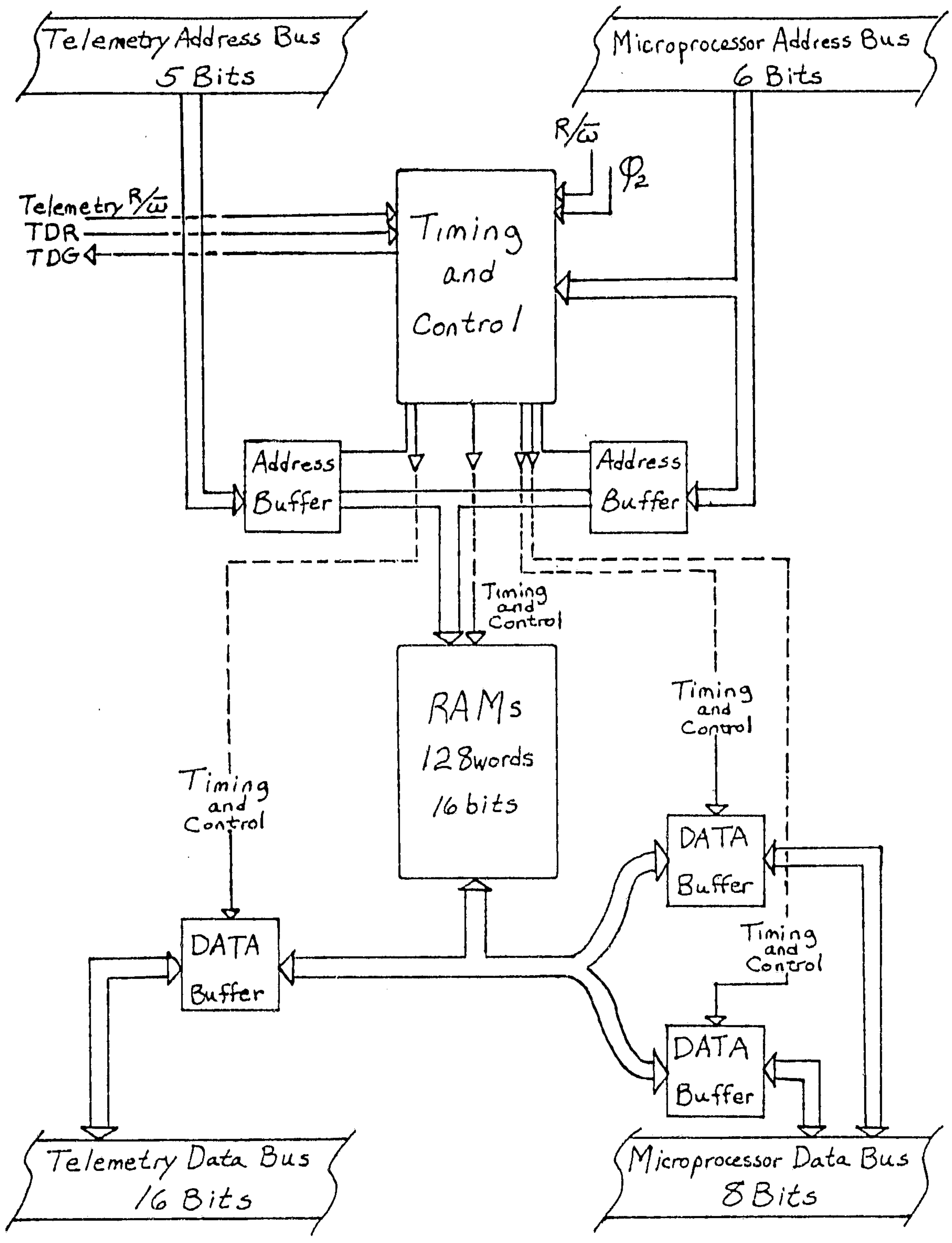


Figure 3-3
Telemetry Interface Block Diagram

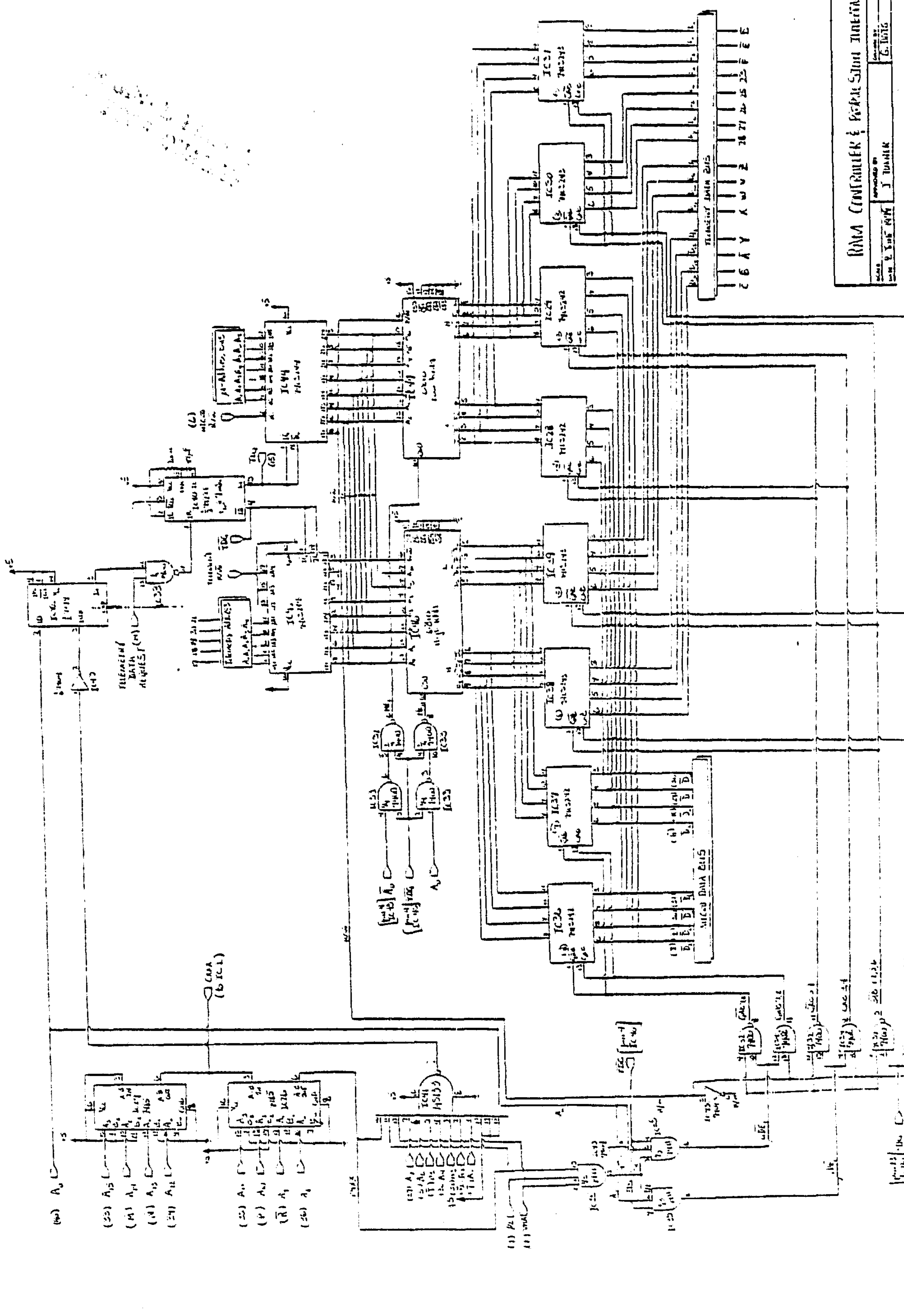
does not receive a TDG during a "Read". TDG can always be ignored during a "Write" since telemetry writes to the RAM for the microprocessor's benefit. The telemetry interface will work with both the old and the new telemetry systems.

The microprocessor obtains sole access to the RAM by writing to a special address (C4FE), and checking to see if the "write" was successful. The "write" clears a flip-flop which disallows new Telemetry Data Requests. Unless telemetry already has sole access to the RAM, the "write" will be successful. The microprocessor retains sole access until it writes to another special address (C4FF) to set the flip-flop. The programmer of the microprocessor must insure that it does not retain access for too long. The telemetry system must have access frequently enough to keep up-to-date vehicle data in the RAM.

Figure 3-4 shows the circuit diagram. The core of the circuit is the memory. Two Motorola MC6810 random access memories (IC46 and IC47) are used in parallel. Around the memories are address and data buffers which give them their own miniature address and data buses. Each memory is organized as 128 8-bit words so that there are a total of 128 16-bit words. Internally, the address bus is 7-bits wide, and the data bus is 16-bits wide.

On the telemetry side of the interface, there are the 16-bit data bus, the 5-bit address bus, and the control

Figure 3-4
Telemetry Interface



signals (TDR, TDG, and Telemetry R/W). TDR is gated through IC33 into the trigger input of a one-shot (IC40). When the microprocessor is not requesting sole access, IC33 is enabled. If TDR goes high indicating telemetry desires access, it triggers IC40 and the output goes high. The output, TDG, is also the main control signal. When TDG goes high, it disables the address and data buffers to the microprocessor and enables the buffers to telemetry. Both memories are enabled so a 16-bit word can be transferred. Telemetry R/W controls the data direction in the data bus drivers and in the RAMs. When TDR goes low again, TDG is cleared.

On the microprocessor side of the interface, two 4-bit magnitude comparitors (IC26, IC27) decode the upper 8 bits of the microprocessor's 16-bit address. (Presently, the RAMs are assigned to C400 through C4FF). A NAND-gate (IC41) matches the bits A7 through A1. When the microprocessor references RAM location C4FE or C4FF, the output of the NAND-gate goes low and clocks address bit A0 into the flip-flop (IC42). Thus, a reference to C4FE clears the flip-flop, and a reference to C4FF sets the flip-flop. When the flip-flop is cleared, it disables IC33 to disallow new TDRs. Note that TDG is not directly affected. The microprocessor must wait until telemetry has finished to actually obtain access. It will continue to write to and read from C4FE until access has been granted. Of course, the microprocessor might actually have access to the RAM before perfoming this procedure, but there is no guarantee unless IC33 disables TDRs first.

The data stored in the RAMs is 16 bits wide, but the M6800 data bus is only 8 bits wide. The least significant bit, A0, selects only one of the RAMs and puts only one byte onto the M6800 data bus. The data buffers from both RAMs are tied to the bus. The "high RAM", selected by an odd address (A0=1), contains the upper byte of the telemetry data word. The "low RAM", selected by an even address (A0=0), contains the lower byte of the telemetry data word. Note that the M6800 data bus is inverted, so the bus drivers (IC28, IC29, IC32, and IC33) must invert.

3.2.3. Motor Driver Interface

The microprocessor compares the wheel set speed to the actual wheel speed and produces a number. This number, the drive command, represents the voltage required by each motor. Unfortunately, a number will not turn the motors. The motor driver interface converts the number into the power the motors require.

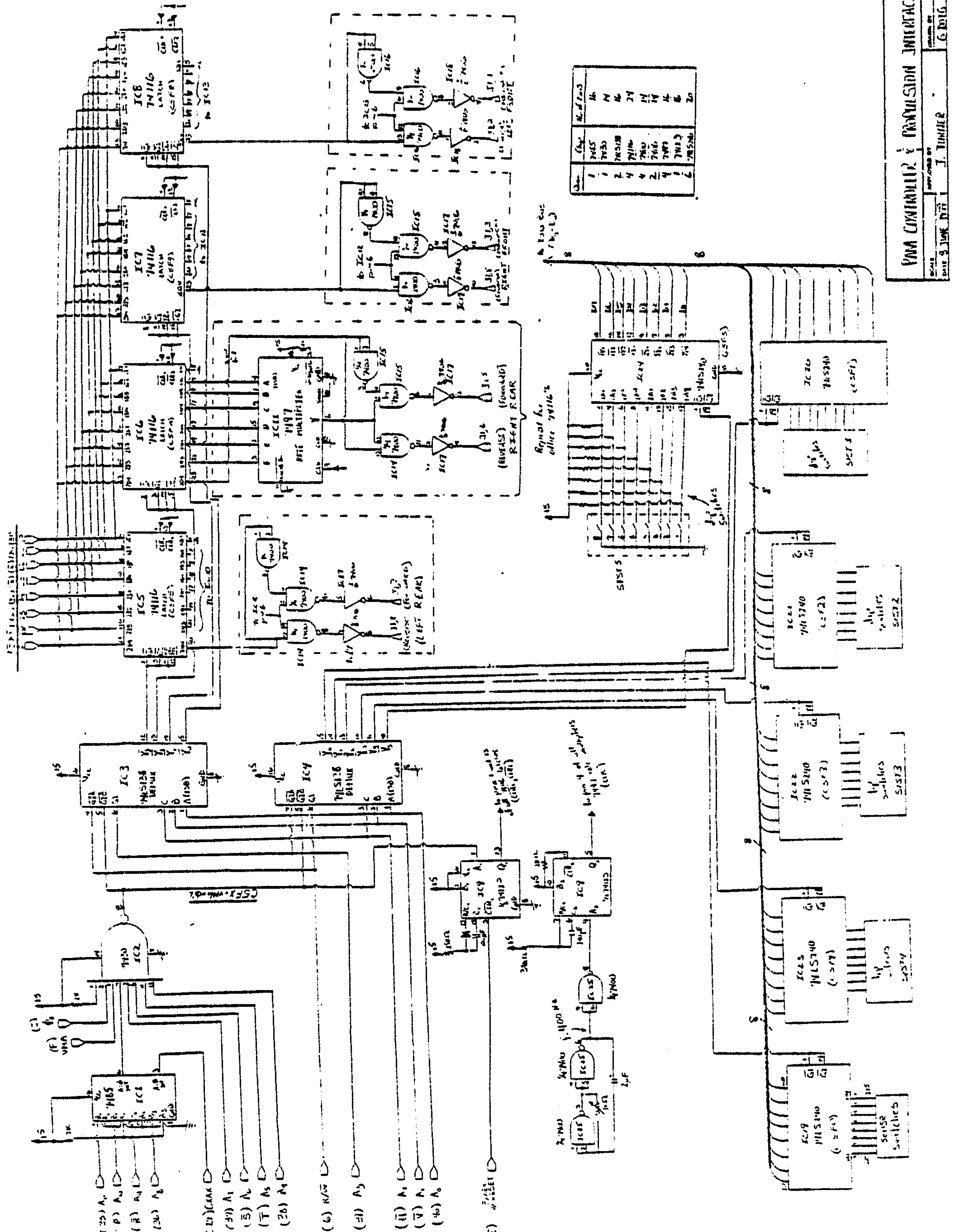
One common, and easy, way to control the power delivered to a DC motor is pulse-width modulation. Each motor is connected to a switching circuit which can turn the supply voltage on and off. If the supply is turned on and off rapidly enough, the motors will filter the voltage and extract the DC component. A control signal determines the length of time that the switch is on. The control signal is a constant frequency, variable duty cycle waveform. The "on" time of the signal determines the "on" time of the switch.

The old propulsion system used pulse-width modulation[3], so the new system uses the same motor driver circuits to switch the 24 volt supply. Thus, the motor driver interface need only supply the control signals. I decided, after experimentation, to use a 400 Hz control signal as was used in the old system. At a lower frequency, the motors begin to jerk. At higher frequencies, the motors hum, heat, and slow down. A frequency around 400 Hz makes the best compromise between filtering and inductive losses in the motor.

All that remains is to convert the drive command number into the appropriate duty cycle. The most straightforward method, to turn a single bit on and off under software control, does not seem to be the simplest in this case. The 400 Hz waveform has a period of 2.5 milliseconds. Thus, the bit may be on from 0 to 2.5 milliseconds. It would be difficult to insure that the proper duty cycle was produced during program operation due to interrupts for vehicle commands and waits for telemetry to finish with the interface RAM. This does not rule out the software approach. Some hardware/software tradeoffs were necessary in the early design stage, and a small amount of hardware does simplify the control algorithms.

Therefore, the variable duty cycle waveform is produced by a rate multiplier (7497)[8]. See figure 3-5. The rate multiplier contains a 6-bit binary counter, which counts 64 clock pulses. A 6-bit binary number (from 0 to 63) is applied to the rate inputs. This number specifies how many pulses to output during each full cycle of 64 clock pulses. That is, if the number 27 was applied to the rate inputs, 27 pulses would be produced each cycle. A pulse is output after each clock pulse, while the clock is low. Therefore, if the clock is low for all but a very short

Figure 3-5
Motor Driver Interface



time, many of the output pulses run together. The rate input specifies the duty cycle rather than the number of pulses. This way, the duty cycle can be controlled to one part in 64.

The microprocessor sets the rate inputs by storing the drive command into an 8-bit latch, which appears as a write-only register. (The write-only registers occupy addresses C5F8 through C5FF). The drive command is in signed binary form: seven bits of magnitude plus a sign bit. Six bits of the magnitude become the rate inputs. The sign bit specifies drive direction. The least significant bit of the drive command should be 0, and is ignored.

The motor drivers can turn the 24 volt supply to the motor on or off, forward or reverse. One control line specifies forward drive and supplies 24 volts to the motor. The other line supplies -24 volts to the motor for reverse drive. The sign bit from the drive command switches the output from the rate multiplier between the control lines through two NAND-gates. High-voltage open-collector drivers act as buffers to the control lines.

There is a latch, a rate multiplier, and a direction switching circuit for each wheel. A common clock feeds all four rate multipliers.

The clock is derived from a 400 Hz square wave, produced by IC25. IC25 drives a monostable multivibrator

(IC9B) which produces very short clock pulses. Since the clock pulses are so much shorter than the clock period, the rate multipliers can approach 100% duty cycle.

Another multivibrator (IC9A), acts as a fail-safe device. Its output is tied to the clear input of all four latches. Every time the microprocessor updates one of the latches with a new drive command, the multivibrator is retriggered for about another 300 milliseconds. As long as the output remains high, the latches retain the drive command. If the program fails to operate properly and the latches are not updated, then the latches will be cleared and the motors turned off. The latches are also cleared upon system restart.

3.2.4 System Parameter Interface

The propulsion and steering system must be easy to calibrate. Therefore, five 8-bit switches are provided on the board. Each switch is assigned an address and appears as a read-only register to the microprocessor. The section 3.3.4 describes how the software interprets the data entered on the switches.

Up to eight read-only registers are possible with the current hardware, but only six are used. They are assigned addresses from C5F0 to C5F7. The five parameter selection switches occupy addresses C5F1 through C5F5. The wheel mismatch sense switches are assigned to address C5F0. All switches are buffered through 3-state drivers onto the microprocessor data bus.

3.3. Control Algorithms

This section describes the algorithms used in the propulsion control system. Since the system is microprocessor based, the algorithms are implemented in software form.

The control program is divided into twelve functional subroutines. It occupies just over 2K bytes of memory.

3.3.1. Initialization

When the power is turned on, the microprocessor automatically issues a "Restart" interrupt. At "Restart", program execution begins at the address contained in memory locations FFFE/FFFF. The propulsion system initialization routine, INIT, lies at this address.

INIT performs five basic tasks. It must clear all propulsion system workspace and driver interface registers, put a copy of the main program into system workspace, initialize the stack pointer, initialize the command receiver and MINIBUG ACIAs, and initialize the relay control PIA. After initialization is complete, INIT transfers control to the program main loop.

The command ACIA (Asyncronous Communications Interface Adapter)[9] is set to accept seven bit words with even parity, and two stop bits. Whenever a word is received, it issues an interrupt request to the microprocessor. Program control would be transfered to the new command processing routine, NEWCMD.

The MINIBUG ACIA transmits characters to the Lear Siegler CRT terminal. The data display routine, DISPLY, uses it during execution of the propulsion system. However, if the MINIBUG ACIA receives any character from the terminal, it will issue a non-maskable interrupt. This transfers control from the propulsion system software to

Motorola's MINIBUG II software[10]. The ACIA accepts the same format as the PRIME computer: eight bit words with no parity bit and two stop bits.

3.3.2. Main Control Loop

Figure 3-6 shows the program top level. The program gathers vehicle state data, and commands from the off-board computer. It develops the proper set speeds using steering and terrain correction algorithms. Then, it performs as a discrete time proportional speed control.

The main loop does nothing but call the functional subroutines. It resides in random access (read/write) memory for easy access during debugging. The program runs continuously. Every time around the loop, the program produces a new set of motor drive commands which it stores in the motor driver interface. It completes each loop in about 12 milliseconds, so the wheel speeds are sampled and updated about 80 times per second.

Stability and error requirements determine the sample rate that is actually needed. As the time between samples becomes longer, the forward path gain must be reduced to insure stability. This increases the steady-state error. Since no correction is made to the motor drive

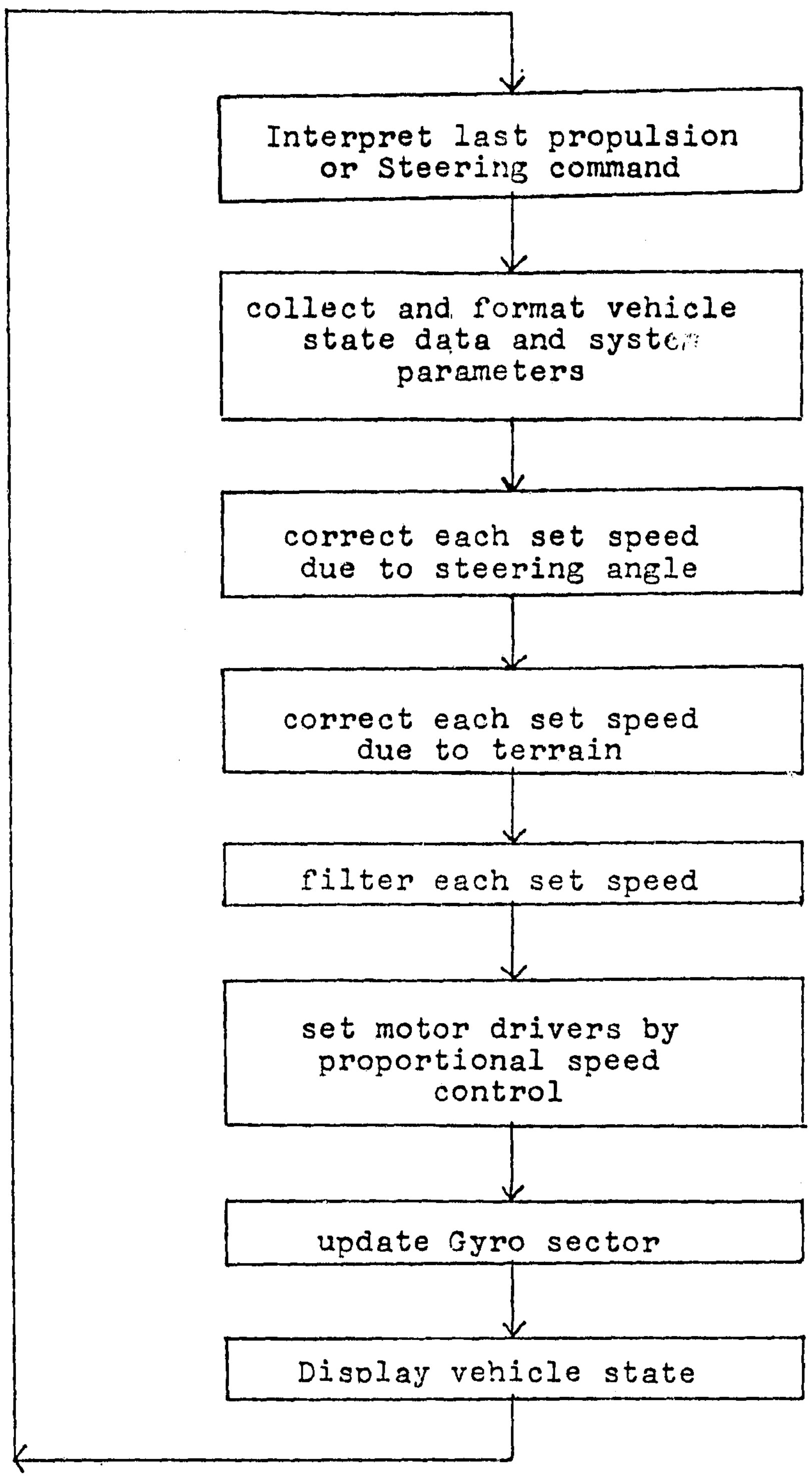

Interpret last propulsion or Steering command
collect and format vehicle state data and system parameters
correct each set speed due to steering angle
correct each set speed due to terrain
filter each set speed
set motor drivers by proportional speed control
update Gyro sector
Display vehicle state

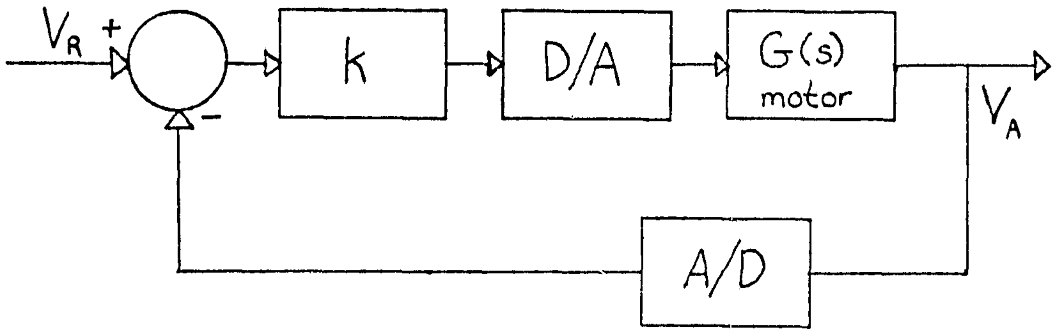
Figure 3-6
Main Control Loop

between samples, errors in the speed can build up. The sample time should be short enough to allow negligible error build-up between samples, and a high forward-path gain to reduce steady-state errors.

Figure 3-7 shows the block diagram of a discrete proportional speed control. If electrical transients are neglected, the motor transfer function, G(s), contains a single pole due to the mechanical transients. The closed-loop discrete transfer function, T(z), also contains a single pole which is dependent upon the forward path gain and the sample time.

Our motors have mechanical time constants of about two seconds under no load. Take two seconds for a conservative estimate. First, consider intersample errors. A rule of thumb is that the sample interval should be one-tenth the time constant of the pole. By this estimate, we can tolerate a sample interval as long as 200 milliseconds. Next, consider stability and steady-state error. Equation 4 in Figure 3-7 gives the steady-state transfer function. Equation 5 gives the upper limit on the forward path gain. For steady state errors under 5%, the gain must be at least 20. This fixes the maximum sample interval at 100 milliseconds.

The sample interval should be smaller that this upper limit for a margin of safety. Our sample interval of



(1) $G(s) = \frac{V_A}{V_R} = \frac{\alpha/JR}{s + B/J + \alpha\beta/JR} = \frac{A}{s+a}$

(2) $G(z) = \frac{A}{a}\left[\frac{1 - e^{-aT}}{z - e^{-aT}}\right]$

(3) $T(z) = \frac{K\,G(z)}{1 + K\,G(z)} = \frac{K'(1 - e^{-aT})}{z - \left[(1+K')e^{-aT} - K'\right]}$

(4) $T(1) = \frac{K'}{1 + K'}$ (steady-state) so $K' = \frac{T(1)}{1-T(1)}$

(5) for stability: $K' < \frac{e^{-aT}}{1 - e^{-aT}}$ or $T < \frac{-1}{a} \ln \frac{K'}{1+K'}$

IF $T(1) = 0.95$, and $a = 0.5$ seconds,

$K' \geq 19$ (gain) $K' = K\frac{A}{a}$

$T \leq 100$ msec (sample interval)

Figure 3-7
Proportional Speed Control

12 milliseconds leaves a wide margin of safety (gains up to 200) and sufficient room for program expansion.

3.3.3. Response to Vehicle Commands

The off-board computer or human operator controls the rover by radio[3]. Vehicle control commands set the rover's speed and steering angle, raise or lower the struts, and initialize the gyro. Two subroutines are responsible for responding to and obeying the commands.

The command port, ACIA U12, issues a program interrupt request (IRQ) to the microprocessor whenever it receives a new vehicle command. Program control is transferred to the routine NEWCMD.

The command ACIA is also connected to the "loss of signal" output from the command receiver. If the command carrier frequency is lost, the "loss of signal" sets an ACIA status bit. NEWCMD checks the status bit. If it is set, NEWCMD stops the wheels. The rover cannot run away uncontrolled.

No attempt has been made to change the command format. It is completely compatible with the old command link. Vehicle commands are seven bit words. The four most significant bits designate a vehicle subsystem. The three least significant bits designate the function which the subsystem should perform. Figure 3-8 gives a list of the commands NEWCMD recognizes.

NEWCMD is short, so that the main program loop execution time is predictable in spite of interrupts. It only partially decodes the vehicle command. In the case of propulsion or steering control commands, it just stores the new command code for the appropriate subsystem. The command is interpreted further and obeyed by GETCMD during normal program execution. Similiarly, a command to initialize the gyro simply updates the gyro status word. The GYRO routine does all of the initialization. Only the strut commands, which are infrequent and simple to obey, are executed by NEWCMD.

The propulsion and steering system itself recognizes four types of vehicle commands. These are the steering commands, main drive commands, front wheel drive commands, and one wheel drive commands.

The old steering system had 15 possible steering positions, so the command link allows four bits to designate the steering angle. One bit of the subsystem code designates whether the steering angle is "left" or "right". The three data bits are a code representing the magnitude of the steering angle. There are seven steering angles in each direction. Of course, there is no reason the new steering

Command Name		Command Code (octal)
One Wheel Drive:	OFF	010
	Left Rear	014
	Right Rear	015
	Left Front	016
	Right Front	017
Steering: (left)	-90.0°	107
	-77.1°	106
	-64.3°	105
	-51.4°	104
	-38.6°	103
	-25.7°	102
	-12.9°	101
	0.0°	100
(right)	12.9°	111
	25.7°	112
	38.6°	113
	51.4°	114
	64.3°	115
	77.1°	116
	90.0°	117
Main Drive:	Forward Speed #3	123
	Forward Speed #2	122
	Forward Speed #1	121
	STOP	120
	Reverse Speed #1	125
	Reverse Speed #2	126
	Reverse Speed #3	127
Strut Control:	Rear Up	152
	Rear Stop	150
	Rear Down	153
	Front Up	157
	Front Stop	154
	Front Down	156
Front Wheel Drive:	Forward	172
	STOP	170
	Reverse	173
Gyro Initialize		167
Display ON		166
Display OFF		164

Figure 3-8
Vehicle Commands Recognized by NEWCMD

system must be limited to 15 steering positions when 256 are readily available. However, the path selection algorithms do not now require more than 15 positions. The addition of more steering positions requires a modification of the command link format.

The main drive command controls the movement of the rover. All four motors are used. The command link allows a stop command, three forward speeds, and three reverse speeds. Only one of these, the slowest forward speed, is used during autonomous roving.

There are two other commands to control the wheels. The front wheel drive command allows the two front wheels to move alone. The one wheel drive command allows only one wheel to run. These commands are not used during autonomous roving. They are included primarily for compatibility with the old command link. As implemented now, the one wheel drive command may help to calibrate the wheel speeds. The selected wheel is set to turn at 16 2/3 RPM. Thus, an ordinary stroboscopic disk, used to check turntable speed, will appear to stand still under indoor lighting when the speed is correct.

The microcomputer responds to three miscellanous commands. One command turns the propulsion display on and off. NEWCMD sets a flag which the routine DISPLY interprets. DISPLY writes the actual steering angles and wheel speeds on a terminal in readable format. Another command sets a flag used by the routine GYRO. The GYRO software insures that the directional gyroscope stays within its linear range. When the flag is set, the gyro initialization procedure begins. The last command controls the position of the front and rear struts. NEWCMD sets the bits that control the strut motors in a PIA. These features are discussed further in the descriptions of DISPLY, GYRO, and Appendix 6.2.

There are plans to add new commands to the command link to control the new laser mast. The off-board computer could specify the center of scan position or even change the laser firing pattern. Some changes in the command format will be necessary to add these commands. I believe that the entire format should be changed at that time. NEWCMD can be modified easily to accomodate this change, and one of the spare PIAs could interface with the laser mast controller. GETCMD need only be changed if the steering angles or wheel speeds are specified by different function codes.

3.3.4. Data and Parameters

The routine GETDAT obtains and formats the vehicle state data and switch-selectable parameters for the propulsion and steering system. GETDAT is the software which drives the Telemetry Interface and System Parameter Interface circuitry.

First, consider the Telemetry Interface. The propulsion and steering system needs the steering angle and each wheel speed. GETDAT obtains these from the telemetry data multiplexer via the Telemetry Interface. The off-board computer needs the directional gyroscope reading and the last command received by the rover. GETDAT sends this information to the telemetry data multiplexer via the Telemetry Interface. All of the data in the Telemetry Interface RAM is 16 bits long.

The steering angle consists of 10 significant bits in two's complement form from the steering shaft encoder. The remaining 6 bits should be zeros. GETDAT rounds to take only the most significant 8 bits. The result, in STeering Units (STU), is used by the steering correction algorithm.

Similarly, the wheel speeds consist of 12 bits in two's complement form from the telemetry A/D converter. GETDAT rounds to take only the most significant 8 bits. SPeed Units (SPU) are used internally. The tachometer gains must be calibrated to give the maximum output from the A/D converter at 0.5 meters/second for a result in SPU. See Appendix 6.3.

The directional gyroscope reading is different. As with the wheel speeds, the telemetry system samples the voltage on the gyroscope potentiometer and produces a 12 bit number. The telemetry interface stores the number at "GYDAT". GYDAT does not identify the direction alone. A four bit segment number, supplied by the gyroscope controller (see section 3.3.8), is also required. The gyroscope controller forms a 16 bit word consisting of the potentiometer reading the the upper 12 bits, and the segment number in the lower four bits. When the routine GYRO performs the gyroscope control function, GETDAT stores the 16 bit gyroscope direction at "GYOUT". The telemetry system should include GYOUT in its polling sequence. It should transmit GYOUT, but not GYDAT, to the off-board computer.

The command echo contains the command receiver status word in the upper byte, and the actual command received in the lower byte. The off-board computer can use this information to determine whether the last command was received properly, and to determine if the command receiver indicates a "loss of signal" from the radio link.

Figure 3-9 shows the addresses assumed by GETDAT for compatiblilty with the old telemetry system. (Note that with the old telemetry system, the gyroscope controller

Data Used by Control System	Microprocessor Data Address	Telemetry Data Address
Command Echo	0, 1	0
Steering Angle	2, 3	1
Left Rear Speed	4, 5	2
Right Rear Speed	6, 7	3
Right Front Speed	24, 25	12
Left Front Speed	28, 29	14
Gyro pot reading (from Telemetry)	30, 31	15
Gyro reading (to Telemetry)	32, 33	16

(Note: the microprocessor addresses are relative to the beginning of the telemetry interface RAM).

Figure 3-9
Telemetry Interface Data Addresses

board, not the GYRO routine, supplied the segment number. Only the first sixteen telemetry words are used). The new telemetry system must use the addresses in figure 3-9, or the data definitions in GETDAT must be changed.

Now consider the switch-selectable parameters. Each parameter is 4 bits, so there are two parameters to each 8-bit word. Figure 3-10 shows the layout of the parameters.

There are five 8-bit DIP switches. Thus, ten 4-bit parameters are available. Only nine are used now. The switches allow one to define each of the three vehicle speeds, the speed controller gains, the time constant of the digital low pass filter, and the rate of turn during steering.

GETDAT is called each time around the main loop. It obtains and formats all of the data each time.

bit	7 ↓ … 4 ↓	3 ↓ … 0 ↓
DIP Switch #1:	Speed #1	Speed #2
DIP Switch #2:	Speed #3	LF Gain
DIP Switch #3:	RF Gain	RR Gain
DIP Switch #4:	LR Gain	Time Constant
DIP Switch #5:	Turn Rate	Spare

The Value of each item is an integer between 0 and 15

Speeds: $0.1 \leq \text{Speed} \leq 0.35$ meters/second

Speed (in SPU) = 25 + 4 x Value

Gains: $0 \leq \text{Gain} \leq 60$

Gain = 4 x Value

Time Constant: $0.875 \leq \text{tau} \leq 0.9921875$

tau = (112 + Value) / 128

Turn Rate: $0 \leq \text{Rate} \leq 0.234$ meters/second

Rate (in SPU) = 4 x Value

Figure 3-10
Switch Selectable Parameters

3.3.5. Wheel Coordination

As discussed in Chapter 2, the propulsion control system should drive the wheels cooperatively at all times. Wheel mismatch feedback enables the two-wheeled model to adjust to irregular terrain. The actual rover is a little more complicated, since there are four wheels.

Mismatch feedback, alone, is insufficient to coordinate all four wheels since the rover uses "wagon steering". See figure 2-1. The entire front axle turns. During strong turns, the rover pivots at a point along the rear axle. Thus, one of the rear wheels must reverse direction. As long as the set speed remains forward, the mismatch controller will assume that the wheel is pushing against an unusually heavy load; it is not dragging. Therefore, the controller must use the steering angle at least to determine the direction the wheel should turn. Actually, the controller uses the equations in figure 2-1 to set the speeds to their proper values on level ground. This enables the controller to work tolerably well even if the mismatch sensors fail.

First, it is necessary to know the rover's desired speed. GETCMD interprets the commands from the operator or off-board computer. GETDAT finds the vehicle speed which corresponds to the speed command code. Then, the routines STRCOR, TERCOR, and FLTR develop the set speeds for each wheel.

STRCOR determines the set speed for each wheel on level ground as a function of steering angle. It uses the equations in figure 2-1, as did the old propulsion system. However, STRCOR uses 8-bit precision rather than 4-bit precision.

STRCOR also turns the front axle if the desired steering angle is not equal to the actual steering angle. The speed of one front wheel is increased, while the speed of the other wheel is decreased by the same amount. Within about 14° of the desired steering angle, the speed increment is about 1 cm/sec for each degree of steering error. Outside of this proportional band, the rate of turn is specified by the "rate of turn" DIP switch.

TERCOR checks the sense switches on the worm gears to see if a correction due to irregular terrain (or even tachometer miscalibration) is required. When the sense switches show that a wheel is dragging, the set speeds are adjusted by TERCOR according to the algorithm in figure 3-11.

Changes in each of the set speeds are smoothed by the routine FLTR. FLTR implements a first order difference equation, so it acts as a digital low-pass filter. Thus, the rover will behave like the model in section 2.3. When the time constant of the filter is properly set, the speeds

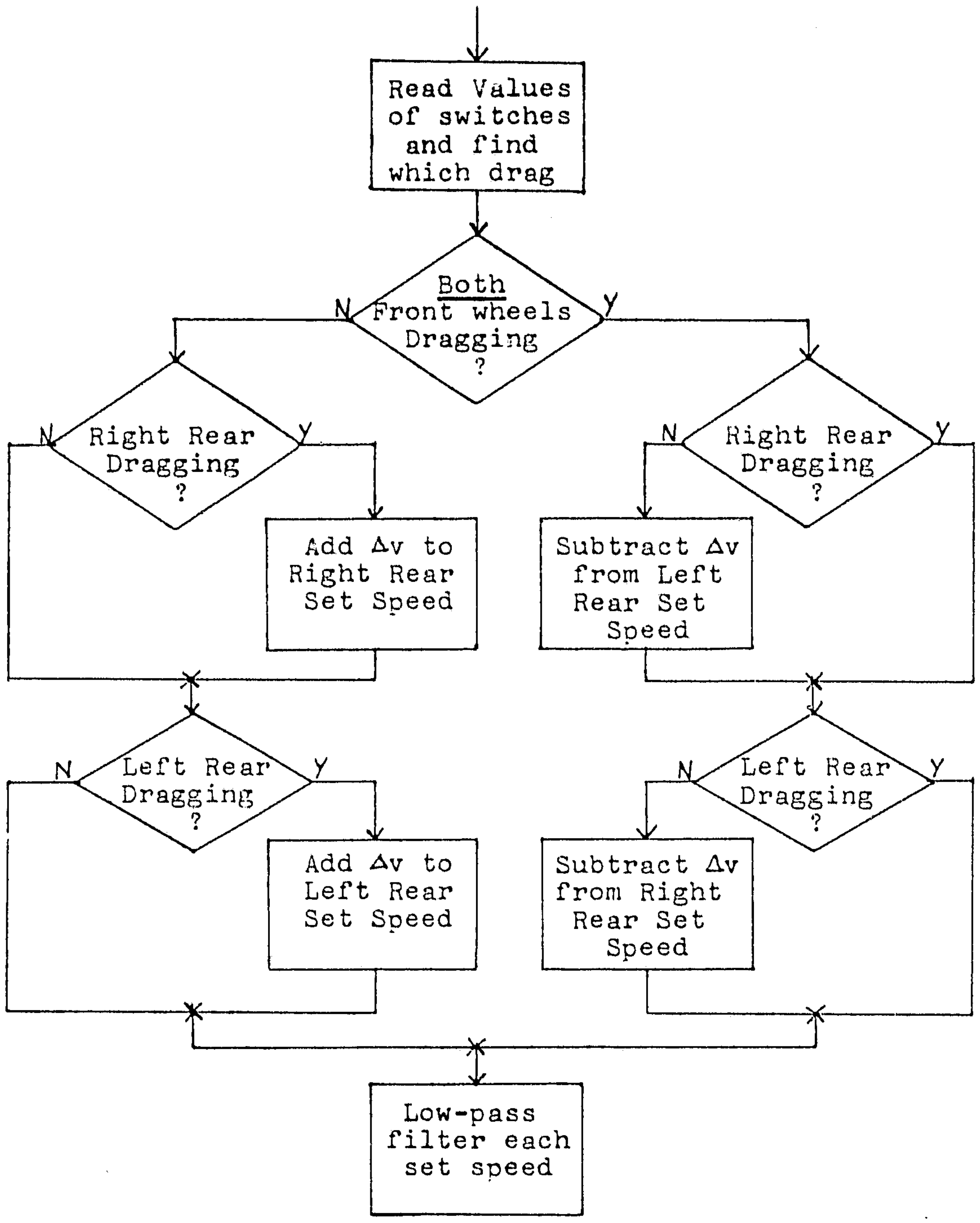

Read Values of switches and find which drag
Both Front wheels Dragging ?
N
Y
Right Rear Dragging ?
N
Y
Add Δv to Right Rear Set Speed
Right Rear Dragging ?
N
Y
Subtract Δv from Left Rear Set Speed
Left Rear Dragging ?
N
Y
Add Δv to Left Rear Set Speed
Left Rear Dragging ?
N
Y
Subtract Δv from Right Rear Set Speed
Low-pass filter each set speed

Figure 3-11
Wheel Mismatch Correction Algorithm

will closely follow the speeds required on the terrain, and the wheels will not fight each another.

3.3.6. Proportional Speed Controller

The routine CONTRL implements four parallel proportional speed controllers:

DRIVE = GAIN * (REFERENCE SPEED - ACTUAL SPEED)

The actual wheel speed is subtracted from the set speed which was found by STRCOR, TERCOR, and FLTR. The difference is multiplied by a constant gain to produce the drive signal for each motor. One copy of the drive signal is kept in system workspace, in two's complement form, for debugging purposes. The other copy is rounded to 7 bits in signed binary form. This number is stored in the proper driver interface register. It specifies the DC voltage to deliver to the motor.

3.3.7. Data Display

The propulsion and steering system will require periodic calibration checks to insure that the wheel speeds and steering angle are accurate. The data display feature, performed by the routine DISPLY, will help with calibration checks. DISPLY formats the important propulsion systems parameters and writes them to a terminal.

The desired and actual steering angles are displayed in decimal degrees. The desired and actual wheel speeds are displayed in decimal millimeters/second. The display also indicates whether or not each wheel is dragging.

Even at 9600 baud, it takes the ACIA about 1 millisecond to transmit each character to the terminal. The entire display, including descriptive titles, is almost 300 characters long. Obviously, the propulsion control system cannot wait for the transmission of almost 300 characters each time around the main loop. Thus, DISPLY does not wait for the ACIA to transmit a character.

DISPLY keeps a pointer to the next character. This pointer doubles as a flag. If the pointer is 0, then DISPLY exits without transmitting any characters. To start the display, the pointer is set to any large number beyond the end of the message text (hex FFFF, for instance). DISPLY will convert each variable from its internal format to its decimal ASCII representation. Then, it will begin to store characters in the ACIA data buffer. The ACIA can hold two characters: one that is being transmitted, and one to transmit next. As soon as the ACIA buffer is full, DISPLY exits.

The message text consists of the data, and descriptive titles to identify the data. These are intermixed at the terminal, but not in memory. The titles reside in read-only memory, whereas the data must reside in read/write memory. In order to simplify the program, a special pointer in read-only memory designates a reference to read/write memory. DISPLY reads each character, in turn, out of read-only memory. Normally, it stores the character into the ACIA data buffer. However, if the most significant bit of the character is on, then it cannot be an ASCII character. Instead, it is a pointer to the next data character. DISPLY extracts an index to the data from the pointer, and gets it from read/write memory.

DISPLY contains five internal subroutines. It calls the routine CONVRT whenever the last text character has been transmitted. CONVRT updates the data characters with the current vehicle state. CONVRT calls DISSTU and DISSPU. DISSTU converts from the internal steering representation, STeering Units, to decimal degrees in ASCII characters. DISSPU converts from SPeed Units to decimal millimeters/second in ASCII. Both of these call CVD and CVA. CVD converts from hexadecimal to decimal. Likewise, CVA converts from decimal to ASCII.

3.3.8. Directional Gyroscope Controller

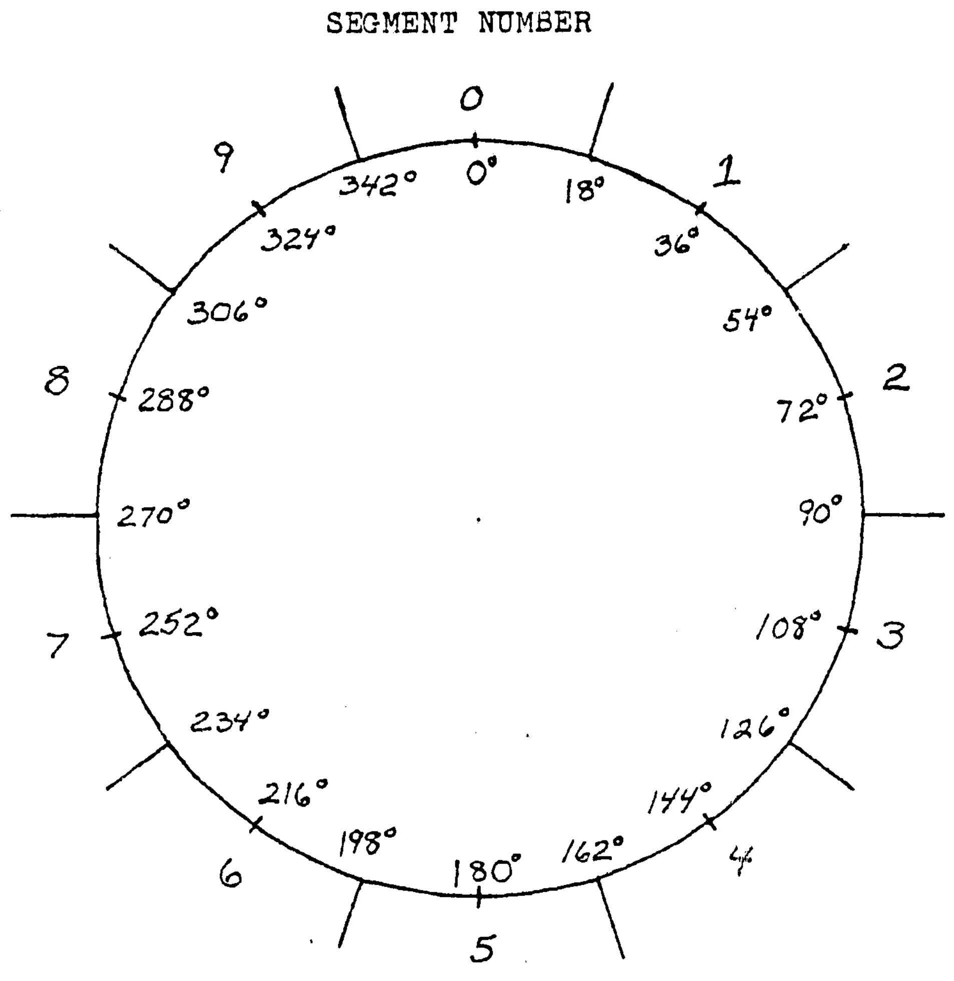
In order to reduce the hardware on board the rover, the directional gyroscope controller board^3 has been replaced by a subroutine. The routine GYRO will be called as part of the main control loop.

GYRO will perform the same function as the hardware it replaces. There are two reasons GYRO has been written. First, it reduces the hardware complexity on board. With GYRO, and the strut control feature in NEWCMD, the microprocessor replaces all of the vehicle control functions. The new telemety system will reside in the microprocessor card cage. Therefore, GYRO eliminates the need for two card cages, and reduces power consumption. Secondly, it simplifies the new telemetry system somewhat. The old telemetry system had a special provision to append the 4-bit segment number from the gyroscope controller to the 12-bit potentiometer reading from the A/D converter. The new telemetry system need not do this. The microprocessor will combine the words and send them to the telemetry system through the interface.

The directional gyroscope on the rover is a Giannini Type 3221. A report by Karen Andersen gives a complete description of it[11]. The gyroscope direction is measured by a potentiometer. Unfortunately, the potentiometer has a linear range of only about 180°. The rover must be able to make turns in excess of 90° in either direction. Fortunately, the gyroscope provides for this. Inside the gyroscope housing are two stepping solenoids. Each can rotate the potentiometer relative to the housing in 2° steps. One solenoid rotates it clockwise, the other rotates it counter-clockwise. The directional gyroscope controller insures that the potentiometer stays within its linear range, and keeps track of the solenoid steps.

The 360° circle is divided into ten 36° segments. Each segment is given a 4-bit binary number, from 0 to 9. The number increases in a clockwise direction. See figure 3-12. Every time the potentiometer passes a certain threshold (about 60° off center) 18 solenoid steps move the potentiometer 36° to the next segment. The segment number is increased or decreased accordingly. A segment number of 15 indicates that the computer should ignore the gyroscope reading since the solenoids are turning the potentiometer.

The gyroscope controller should be initialized at the beginning of each autonomous test. During



Add (36° x Segment Number) to gyroscope reading to obtain the true heading

Figure 3-12
Gyroscope Segment Numbers

initialization, GYRO steps the solenoids until the potentiometer reads about zero volts. Then it sets the segment number to zero. Initialization begins when a "GYRO initialization command" is received, or when the initialization button on the rover is pushed.

Two lines of a PIA (U5B) control the stepping solenoids. Every fourth time around the main loop, the appropriate control line turns a solenoid on or off. This results in a stepping frequency of about 12.5 Hz. The manufacturer recommends a maximum frequency of 15 Hz.

3.3.9. Multiplication

The M6800 microprocessor does not have a multiplication instruction. Therefore, a subroutine was included in the propulsion and steering system to multiply two 8-bit two's complement numbers.

The routine MULT8 produces two results: a 16-bit product, and an 8-bit product. The 8-bit product is the most significant portion of the 16-bit product. Most of the routines use it. MULT8 uses Booth's algorithm and was adapted from the multiply routine given in the Motorola applications manual[12].

4. Discussion and Conclusion

The new propulsion and steering control system should greatly improve the rover's performance on all types of terrain. Even without the mismatch sensors on the worm gears, the wheel speeds will be consistent and accurate on level ground.

The microprocessor also performs control functions such as vehicle command interpretation, strut control, and directional gyroscope control. Of course, the microprocessor can receive and interpret commands to control new systems with little effort. For instance, the laser firing pattern and center of scan might be specified by the off-board computer and changed by the microprocessor.

There is still room for improvement. Some features are not yet implemented.

The control system should impose a torque limit on each motor, so that they will not damage the gears and wheels if they become stuck. The motors have damaged the rover in the past.

If the need for tachometer calibration still poses a problem to navigation, the tachometers should be replaced by incremental optical encoders. These are expensive commercially but can be built rather cheaply.

It would be interesting to have the microprocessor check the attitude gyroscope. If the rover reaches a precarious pitch or roll, the propulsion system would automatically shut down to prevent the rover from overturning.

The wheel set speeds are low-pass filtered to produce a smooth transition from start to stop, and to enable the wheel speeds to follow those required by irregular terrain. Since the set speeds drop off so slowly, steering overshoot might become a problem. If it does, then the whole set speed should not be filtered. The speed increment added by terrain compensation could be filtered separately. Some other method could be used to create a reasonable velocity profile during starts and stops.

As of this writing, all of the system hardware and software has been checked separately. The software produces the correct results on the M6800 emulator. The microprocessor and the interfaces seem to function correctly. The system has not yet been tested with the other rover subsystems. It should be fully integrated into the rover by the Fall of 1979.

5. Literature Cited

1. Cairns, Steven,
"Terrain Sensing System (Laser Transmitter and Data Handler/Controller)", Rensselaer Polytechnic Institute, Troy, N.Y., May, 1976.

2. Craig, John,
"Elevation Scanning Laser/Multi-Sensor Hazard Detection System", Rensselaer Polytechnic Institute, Troy, N.Y., August, 1978.

3. Geis, Timothy R.,
"Control Electronics for the Mars Roving Vehicle", Rensselaer Polytechnic Institute, Troy, N.Y., May, 1976.

4. Knaub, David,
"Evaluation of the Propulsion Control System of a Planetary Rover and Design of a Mast for an Elevation Scanning Laser/Multi-Detector System", Rensselaer Polytechnic Institute, Troy, N.Y., July, 1978.

5. Motorola,
M68ADS2A Development System User's Guide, Second Edition, Motorola, Inc., 1979.

6. Fell, Kathering A.,
"The Telemetry Link and Computer Interface Designed for the Mars Rover", Rensselaer Polytechnic Institute, Troy, N.Y., December, 1975.

7. Cipolle, David J.,
"Telemetry Data Link System", Rensselaer Polytechnic Institute, Troy, N.Y., May, 1979.

8. Texas Instruments, Inc.,
The TTL Data Book for Design Engineers, Second Edition, Texas Instruments, Inc., 1973.

9. Motorola,
The Complete Motorola Microcomputer Data Library, Motorola, Inc., 1978.

10. Motorola,
Minibug II Source Listing, Motorola, Inc., 1974.

11. Andersen, Karen E.,
"Mars Rover Gyroscope System" Rensselaer Polytechnic Institute, Troy, N.Y., May, 1979.

12. Motorola,
M6800 Microprocessor Applications Manual, Motorola, Inc., 1975.